Grant Agreement No.: 317814

IRATI

Investigating RINA as an Alternative to TCP/IP

Instrument: ***Collaborative Project***
Thematic Priority: ***FP7-ICT-2011-8***

D3.1 First phase integrated RINA prototype over Ethernet for a UNIX-like OS

Due date of the report: Month 11
Actual date: 18th December, 2013
Start date of project: January 1st, 2013 - Duration: 24 months
version: v.1.0

Project co-funded by the European Commission in the 7th Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	✓
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

FP7 Grant Agreement No.	317814
Project Name	Investigating RINA as an Alternative to TCP/IP
Document Name	IRATI D3.1
Document Title	First phase integrated RINA prototype over Ethernet for a UNIX-like OS
Workpackage	WP3
Authors	Francesco Salvestrini (Nextworks) Nicola Ciulli (Nextworks) Eduard Grasa (i2CAT) Miquel Tarzan (i2CAT) Leonardo Bergesio (i2CAT) Sander Vrijders (iMinds) Dimitri Staessens (iMinds)
Editor	Francesco Salvestrini (Nextworks)
Reviewers	Eduard Grasa (i2CAT), Dimitri Staessens (iMinds)
Delivery Date	18th December 2013
Version	V1.0

Abstract

This deliverable presents the first phase integrated prototype of the IRATI project. The prototype implements core parts of a RINA stack over Ethernet for a Linux-base OS, spans both kernel and user spaces and provides different software packages implementing the various functionalities.

The kernel space components mainly lay on the fast-path, implementing the forwarding functionalities through the use of EFCP, RMT, PDU Forwarding table, normal and shim IPC Processes (i.e. the shim IPC over Ethernet) components. These components are bound together by the KIPCM (the kernel IPC manager), the KFA (the kernel flow allocation manager) and the RNL (the Netlink manager) layers that also implement the kernel/user interface.

The kernel part of the IRATI stack must jointly work with its counterpart in user space which provides the remaining functionalities through a well defined set of user-space libraries and OS processes.

The libraries wrap the kernel-space APIs (syscalls and Netlink messages) and provide additional functionalities such as:

- Allow applications to use RINA natively, enabling them to allocate and deallocate flows, read and write SDUs to these flows, and register/unregister to one or more DIFs.
- Facilitate the IPC Manager to perform the tasks related to IPC Process creation, deletion and configuration.
- Allow the IPC Process to configure the PDU forwarding table, to create and delete EFCP instances, to request the allocation of kernel resources to support a flow etc.

These C/C++ based libraries allow IRATI adopters to develop native RINA applications. Language bindings to interpreted languages (i.e. Java) are also made available by wrapping the exported symbols with the target language native interface (i.e. JNI).

Upon these bindings OS daemons such has the IPC Process and the IPC Manager have been developed and are ready to be used for testing and experimentation purposes.

The prototype provides frameworks for configuration, building and development tasks. These frameworks follow common and well established practices with particular emphasis on usability features. The prototype configuration framework automatically adapts to different OS/Linux based systems, e.g. Debian/Linux, Fedora, Ubuntu. The building framework allows to build the whole stack unattended. The software development framework provides a RAD environment.

However, the phase 1 prototype has known limitations which are currently being addressed and will be resolved in the next prototypes. They can be summarised as follows:

- Only flows between adjacent IPC Processes are supported, since the implementation of the link-state routing specification provided in D2.1 is planned for phase 2.
- Only unreliable flows are supported, since the Data Transfer Control Protocol (DTCP) is planned for phase 2.
- The prototype uses simple policies, i.e. only basic functional configuration of most components.

This document presents the relevant choices, protocols and methodologies on software design, development, integration and testing agreed within the partners for the scopes of the first phase prototype implementation.

The software meets both the features and the stability requirements for experimentation and has been released to WP4 (MS7). The core functionalities available in the presented prototype will be further enhanced in the upcoming project prototypes which will be made available in the next periods as part of deliverables D3.2 and D3.3.

TABLE OF CONTENTS

LIST OF FIGURES

Acronyms

ARP	Address Resolution Protocol
CDAP	Common Distributed Application Protocol
CEP	Connection End Point
CLI	Command Line Interface
COW	Copy On Write
DIF	Distributed IPC Facility
DTCP	Data Transfer Control Protocol
DTP	Data Transfer Protocol
DVCS	Distributed Version Control System
EFCP	Error and Flow Correction Protocol
FIDM	Flows-IDs Manager
FIFO	First In First Out
FS	File System
GHA	Generic Hardware Address
GPA	Generic Protocol Address
GPB	Google Protocol Buffers
GUI	Graphical User Interface
HW	Hardware
IDD	Inter-DIF Directory
IPC	Inter-Process Communication
IPCP	IPC Process
JAR	Java Archive
JNI	Java Native Interface
JVM	Java Virtual Machine
KFA	Kernel Flow Allocation (Manager)
KIPCM	Kernel IPC Manager
LAN	Local Area Network
LOC	Lines Of Code
MS	Milestone
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NI	Native Interface
NIC	Network Interface Card
NL	Netlink
OO	Object Oriented
OOD	Object Oriented Design
OOP	Object Oriented Programming
OS	Operative System
PDU	Protocol Data Unit

PIM	Port / IPC Process instance Mapping
RFC	Request For Comments
RIB	Resource Information Base
RINARP	RINA ARP (adaptation layer)
RMT	Relay and Multiplexing Task
RNL	RINA Netlink (abstraction) Layer
SCM	Software Configuration Management
SDU	Service Data Unit
SPR	Software Problem Report
SW	Software
SWIG	Software Wrapper and Interface Generator
UI	User Interface
VA	Virtual Appliance
VB	Virtual Box
VLAN	Virtual LAN
VM	Virtual Machine
VMI	Virtual Machine Image

1 Introduction

The software components required for the first phase IRATI project prototype, as described in the high level software architecture deliverable (ref. D2.1), have been designed, developed, integrated and functionally tested in a virtualised environment. The resulting prototype implements the core components of a RINA stack prototype over Ethernet for a Linux-based OS. Although the design and development activities progressed without major problems and deviations, the release of the first phase software prototype was slightly delayed, mainly due to the unsuitability of the Linux ARP implementation for the scope of the Shim Ethernet IPC Process. That problem caused the introduction of unplanned developments, delaying the prototype release date.

The software prototype meets both the feature and stability requirements for experimentation. Its core functionalities will be further enhanced during the next project phases and will be made available in the next periods as part of deliverables D3.2 and D3.3.

This document presents the relevant choices, protocols and methodologies on software design, development, integration and testing agreed among the partners. It also includes installation and operation instructions of the prototype.

The deliverable is structured as follows. Section 2 presents design and development details as well as updates to the high level software architecture introduced to overcome the problems encountered during development. Section 3 presents both the development model agreed among the partners and the release model agreed between WP3 and WP4. Section 4 presents the environments for development, building and testing of both kernel and user space components. Section 5 and 6 provide instructions on installing and loading the IRATI stack respectively. Section 7 provides instructions on performing tests using the whole stack, including the Shim Ethernet IPC Process, in a dual-machine virtualized environment. Finally, section 8 concludes the document, presenting the work planned for the next prototypes.

2 Software design

This section presents the major updates to the high level software design of the IRATI stack components with respect to the design presented in deliverable D2.1, [3], which further details the high level software architecture.

Figure 1 presents the high-level layout of the IRATI stack components which are detailed in the following sections. The Kernel Flow Allocation Manager (KFA), the RINARP and the ARP826 components depicted in the figure were introduced into the IRATI kernel-space software architecture in order to overcome problems discovered during the software integration phase. The KFA takes over part of the KIPCM functionalities described in D2.1 by managing the kernel-space flow-allocation mechanisms and the interactions with the IPC Process (IPCP) instances, mainly solving concurrency related problems. The RINARP and ARP826 components instead, have been introduced to solve limitations of the current Linux ARP implementation.

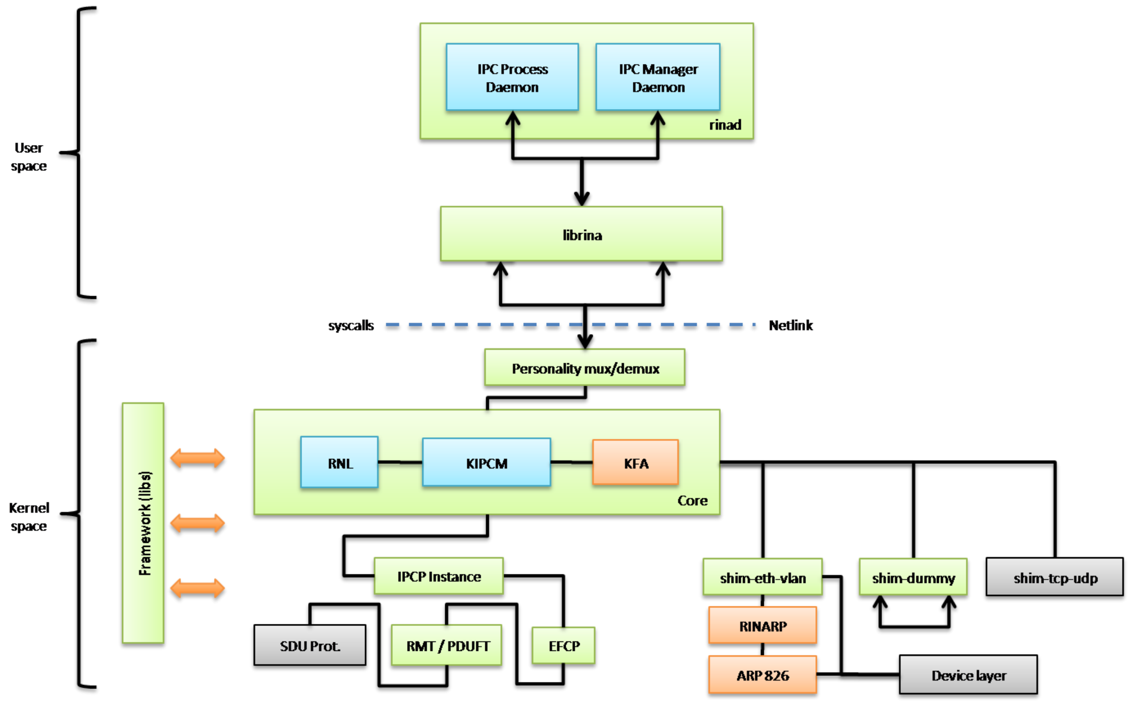

Figure 1 IRATI high-level software architecture

The IRATI stack will continue to evolve during the whole project lifetime and the software design presented in this document will be updated accordingly. The next prototypes may change the design presented in this document significantly. In that case, planned WP3 deliverables D3.2 and D3.3 will report such updates.

2.1 The kernel space

The following subsections cover the design decisions which led to the present kernel-space software architecture as well as the kernel components internals.

Figure 2 provides a detailed picture of the kernel parts referenced in Figure 1.

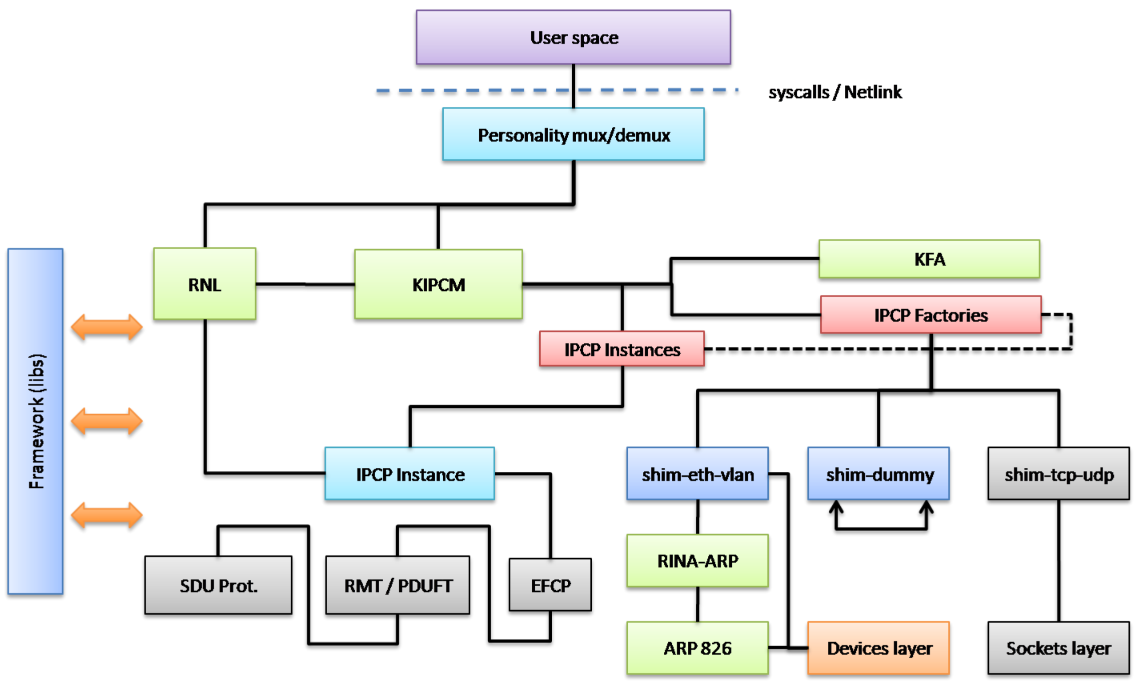

Figure 2 The IRATI kernel-space stack software architecture

2.1.1 The object model

The kernel space software must comply with more restrictions and different requirements compared to the well-known user-space software. As an example, some of the requirements to take care of when designing kernel-space software can be summarised as follows:

- Only a constrained set of programming languages can be used: the kernel-level software must be written either in assembly language or in C. This constraint implies that only a reduced set of Object Oriented (OO) techniques can be applied: there is no way to bind actions to language constructs for implementing ad-hoc object constructors, operators cannot be overloaded etc.
- Kernel contexts have timing constraints: Interrupt handling procedures introduce strict timings that imply programming constraints on the handlers as well as to all the

derived code executing on the same context. Code in such contexts has to react in a timely fashion and cannot call functions that might sleep (such as deferred memory allocation routines).

- Different synchronization and concurrency techniques: The Linux kernel is a cooperative environment where “live” entities can be implemented as tasklets, kthreads, workqueues, etc. The synchronization semantics among these entities differ from user-space implementations through a higher degree of details (e.g. spinlocks, futexes, mutexes, semaphores) and constraints (e.g. spinlocks cannot be recursively locked).

In order to overcome the possible limitations of the environment, reduce the problems that may be caused by incrementally introducing features during the entire project lifetime and keep code refactoring at the minimum, an ad-hoc Object Oriented Design (OOD) approach throughout all the IRATI stack kernel-space components was adopted.

Object Oriented Programming (OOP) approaches applied to low-level languages such as C are not new and lot of literature describing different techniques and methodologies is easily available in the public domain [28]. These approaches however usually deal with OOP in user-space and have to be opportunely re-factored in order to be applied to kernel space. Part of these techniques are already embraced in the Linux kernel (and were applied to the IRATI implementation accordingly) while the remaining ones were introduced during the development phase where needed.

The adopted OOD techniques can be summarised as follows:

- Information hiding: Implemented making use of forward definitions of the object’s type in its header file while the complete definition is embedded into the corresponding compilation module (i.e. using opaque pointers, [29]).
- Constructors and destructors: Implemented making use of ad-hoc and per-object functions mimicking (C++) constructors and destructors. Statically allocated objects are initialised and finalised through the use of _init() and _fini() functions while the dynamically allocated ones are created and destroyed through the use of _create() and _destroy() functions.
- Classes, methods, polymorphism and inheritance: Classes are implemented through the use of opaque data pointers and function pointers stored into the corresponding object, coupled with an ad-hoc API, provide the object’s (public) methods. Polymorphism and inheritance can be easily obtained by opportunely mangling these pointers and enforcing common naming constraints.

By applying such techniques, the internal implementation of a stack component can be hidden and its interface “exported” as a set of function pointers and an opaque data pointer (pointing to the component’s internal state). That interface can be dynamically bound to other components by storing these function and data pointers into the targets, received through the use of an ancillary function.

With the general adoption of the aforementioned approaches throughout the stack and by using the runtime dynamic linking features of the kernel (i.e. module loading and unloading), the implementation of dynamic embedding and removal of features into the stack at runtime is possible. This avoids the need to recompile/change the core of the stack each time the internal implementation of a component changes, which is especially important for the shim IPC processes since they strongly depend on the underlying technology (e.g. Ethernet, WiFi) and thus their implementation varies widely. This way, new technologies can be easily integrated into the IRATI stack.

The following section introduces the current Linux kernel object model and briefly discusses the downsides for its applicability within the IRATI prototype environment, justifying the introduction of a custom object model. The section after that highlights the details of the IRATI object model implementation.

2.1.1.1 The Linux kobjects

The Linux kobjects are an OO abstraction used in the Linux kernel. They initially represented the glue holding the device model and its sysfs interface [32] together. Nowadays their use is widespread over the entire kernel and constitutes the major OOD technique utilised.

The kobject can be summarised as a structure (i.e. struct kobject). Kobjects have a name, a reference count, a parent pointer (allowing kobjects to be arranged into hierarchies), a specific type and (optionally) a representation in the sysfs virtual filesystem. Their representation is the following (as in kernel version v3.10.0):

```
struct kobject {
    const  char *          name;
    struct list_head       entry;
    struct kobject *       parent;
    struct kset *          kset;
    struct kobj_type *     ktype;
    struct sysfs_dirent *  sd;
    struct kref            kref;
    unsigned int           state_initialized        : 1;
    unsigned int           state_in_sysfs           : 1;
    unsigned int           state_add_uevent_sent    : 1;
    unsigned int           state_remove_uevent_sent : 1;
    unsigned int           uevent_suppress          : 1;
};
```

Where, most noticeably:

- **name**: is the symbolic object name, used for both naming (i.e. lookups) and sysfs presentation.
- **entry** and **parent**: are used for objects (re-)parenting.
- **sd**: is used for sysfs presentation (i.e. sysfs directory).
- **kref**: is used for reference counting.
- **ktype**: Represents the type associated with a kobject. It controls what happens when a kobject is no longer referenced and it also drives the kobject's default representation in sysfs.
- **kset**: The kset is the basic container type for collections of kobjects and allows for their homogeneous handling and eases their integration into sysfs (a kset can be represented almost automatically as a sysfs directory, generally each of those entries corresponds to a kobject in the same kset).

Kobjects are generally not interesting on their own; They are usually embedded within some other structure which contains the information the code is really interested in. They can be seen as a top-level abstract class from which the rest of classes are derived in an OO approach. Refer to [27] for further details.

Despite its wide use within the Linux kernel, the kobject abstraction has noticeable downsides when applied to the IRATI implementation: the kobject abstraction makes implicit use of concurrency semantics (i.e. spinlocks), embeds reference counting (i.e. kref), implies objects naming (i.e. the “name” field), forces loose typing etc. The resulting model, initially tailored to hold the bindings of the kernel devices tree, becomes heavy-weighted and cannot be efficiently adapted to the IRATI stack since Its components have different bindings schemas, locking semantics and memory models.

The object model presented in the next section removes the unnecessary details from kobjects while keeping the interesting ones in order to obtain a lightweight and high-performance implementation tailored to the needs of the IRATI prototype.

2.1.1.2 The IRATI objects

As introduced in the high-level software architecture [3], all the kernel-space parts of the IRATI stack have been developed assuming the following conventions in order to keep the code clean and manageable:

- Use of opaque pointers (i.e. `struct obj *` in the functions arguments) to hide the object’ internal representation from its user.
- The object interface exposes functions mimicking constructors (`obj_init()` and `obj_create()`) and destructors (`obj_fini()` and `obj_destroy()`), depending if the objects are statically or dynamically allocated.

- The object interface exposes methods that can be applied over objects of type '`struct obj *`'. When polymorphism is needed, these methods are implemented as function pointers.

The final representation, expressed for a generic object '`obj`', can be summarised with the following interface:

```
struct obj;

int            obj_init(struct obj * o, <parms>);
int            obj_fini(struct obj * o);

struct obj * obj_create(<parms>);
int            obj_destroy(struct obj * o);

int            obj_method_1(struct obj * o, <parms>);
...
int            obj_method_n(struct obj * o, <parms>);
```

2.1.1.2.1 Spinlock, non-interruptible contexts and objects creation

In order to fulfil the kernel constraints, the object-oriented approach summarised in the previous section was further enhanced to be suitable even to non-interruptible contexts.

In general, functions that may sleep cannot be used in non-interruptible contexts, in order to avoid soft-lockups. The problem is mostly noticeable with interruptible memory allocations executed while holding spinlocks. The following code snippet reproduces a typical soft-lockup problem:

```
a_type_t * t;
spinlock_t s;

spin_lock(&s);
...
t = kmalloc(sizeof(*t), GFP_KERNEL);
...
spin_unlock(&s);
...
```

To resolve the aforementioned problem, memory allocation has to be forced as non-interruptible (i.e. the `GFP_KERNEL` flag must be replaced with the `GFP_ATOMIC` one).

Therefore, the previously introduced object creation "method" (i.e. `_create`) has been differentiated at its very root and sliced as follows:

```
struct obj * obj_create_gfp(gfp_t flags, <parms>);
struct obj * obj_create(<parms>);
struct obj * obj_create_ni(<parms>);
```

Where:

- **`obj_create_gfp`**`()`: provides the actual implementation of the object creation, taking as input parameters the flags that control the priority of the memory allocation operations – that is, whether the memory allocation functions can sleep. This function is not (usually) exported in the object' API but declared as "`static`" and hidden into the software module (i.e. the ".c" file).
- **`obj_create`**`()`: has to be used in interruptible contexts. It calls `obj_create_gfp` passing the `GFP_KERNEL` flag (allowing memory allocation operations to sleep).
- **`obj_create_ni`**`()`: must be used in non-interruptible contexts. It calls `obj_create_gfp` passing the `GFP_ATOMIC` flag (preventing memory allocation operations from sleeping).

2.1.2 The framework

All the components of the kernel-space IRATI stack rely on a base framework that is composed by the following parts:

- rmem: This part implements a common memory management layer that provides additional features over the basic primitives available for dynamic memory allocation and de-allocation (i.e. `kmalloc`, `kzalloc` and `kfree`). These features provide additional debugging functionalities such as memory tampering (adding pre-defined shims on-top/at-bottom of the memory area to detect out-of-rage writes) and poisoning (initializing the object contents with a known value to detect uninitialized usage) specifically tailored to RINA objects; allowing developers to easily spot memory leaks as well as memory corruption problems.
- rref: The rref component provides reference-counting functionalities to RINA objects and allows implementing lightweight garbage-collection semantics in a per-object way. Developers can opt-in for reference counting in their objects only when needed.
- rwq, rbmp and rmap: These parts implement façades for the Linux work-queues, bitmaps and hashmaps respectively. These components provide additional functionalities such as easier dynamic allocation, simplified interaction and ad-hoc methods for non-interruptible contexts.

- rqueue: The rqueue component mainly provides dynamically resizable queues, which are unavailable as part of the default kernel libraries.

Part of the aforementioned features, such as memory tampering and poisoning, can be opted-out at compilation time - they are selectable items in the Kconfig configuration framework - and therefore their performance degradation can be reduced to nothing.

The final software framework, used by all the components of the IRATI stack, can be imagined as built on the just presented functionalities and the approach presented in 2.1.1.2. As an example SDU, PDU, KFA and KIPCM internal data structures have been modelled upon this base framework by:

- Including an IRATI object in their representative structures.
- Using the former presented utilities for memory allocation (i.e. rmem) or creation of elements in their structures (rbmp, rmap, rqueue, rwq etc.).

2.1.3 The personalities and the kernel/user interface

The “personality layer” aims to support different implementations of the RINA kernel stack coexisting in the same system. To achieve this, a unique kernel/user interface must be agreed and shared among the different implementations. The kernel/user interface is currently shaped by the set of system calls and the RINA Netlink layer (RNL).

The personality layer acts as a kernel/user interface mux/demux that allows hot-plugging different implementations into the stack core. Each implementation is associated with a personality instance, which is identified by a system-wide unique identifier. In the user-space to kernel direction, the personality layer demuxes the interface to the specific personality instance (e.g. providing the service). In the opposite direction, the personality layer muxes the personality instance to the related user-space entity (e.g. requesting the service). This is possible because the components of the stack core hold bindings to their parent/siblings in a hierarchical way (where the personalities reside at the root of the hierarchy).

2.1.4 The stack core

The components of the IRATI prototype located in kernel space are divided into two main categories: the core and the IPC processes. The core is the set of components that provide the fundamental functionalities to the system, as well as the ‘glue’ that holds together the different instances of IPC Processes. The components in the core orchestrate the interactions with user-space components, the life-cycle of the IPC processes and the data transfer operations (in which the IPC processes are involved).

The core components have been updated since D2.1 [3] resulting in a final composition for the first prototype currently shaped by the KIPCM, the KFA, the RNL and the IPC process factories as depicted in Figure 2. With respect to [3], two main problems were addressed and solved, in addition to minor improvements:

- Concurrency and memory allocation problems.
- IPC process abstraction independent from its type (i.e. normal or shim).

During the testing and integration activities, a few concurrency (i.e. bad locking semantics) and memory (leaks and corruptions) related problems have been detected. Although most of the problems were due to bugs, the coupling between the KIPCM and the IPC Processes has been found too tight.

This coupling and the concurrency constraints imposed by the kernel environment - such as the absence of recursive locking semantics - caused dead-locks under particular read/write conditions. In order to properly solve them, some of the KIPCM functionalities have been moved into a new software component: the Kernel Flow Allocation Manager (KFA).

2.1.4.1 KFA, the Kernel Flow Allocation Manger

The KFA is in charge of flow management related functionalities such as creation, binding to IPC processes, de-allocation, etc. These functionalities are offered to both the KIPCM via its northbound interface, and to the IPC Processes via its southbound interface.

The management of port identifiers and its association to flow instances is the most important functionality provided by the KFA through its Port ID Manager (PIDM). This solves synchronization issues during the flow creation process due to race conditions. The destination applications started to invoke system calls on port-ids before the kernel completed the flow allocation process. Thus, there was a short period where the read and write operations on this port id failed since this situation was not properly handled by the KIPCM. With the introduction of the KFA, the internal kernel flow structures that support the port-ids are created before notifying the application of an incoming flow allocation request, with the status of '`PENDING`'. When the application confirms that the flow is accepted, the status of the flow structure is updated to '`ALLOCATED`'. If the application invokes a read/write system call on the port-id before the flow status is '`ALLOCATED`', the KFA simply deschedule the calling process until the flow allocation is complete.

Complementary to the port id management, the KFA binds the KIPCM and IPC Processes by means of the flow it manages. For this reason, the KFA creates a flow structure for each flow and binds together its port identifier and the identifier of the IPC Process supporting the flow. The flow structure also contains information such as the flow instance allocation state – as described in the previous paragraph - and the queue of SDUs ready to be consumed by user space applications. By means of the allocation state, the KFA controls the life cycle of a flow, since the possible states (pending, allocated, de-allocated) are set as result of the management actions and impose constraints about the possible actions that can be applied over the flow.

The following figure shows the main interactions of the KFA.

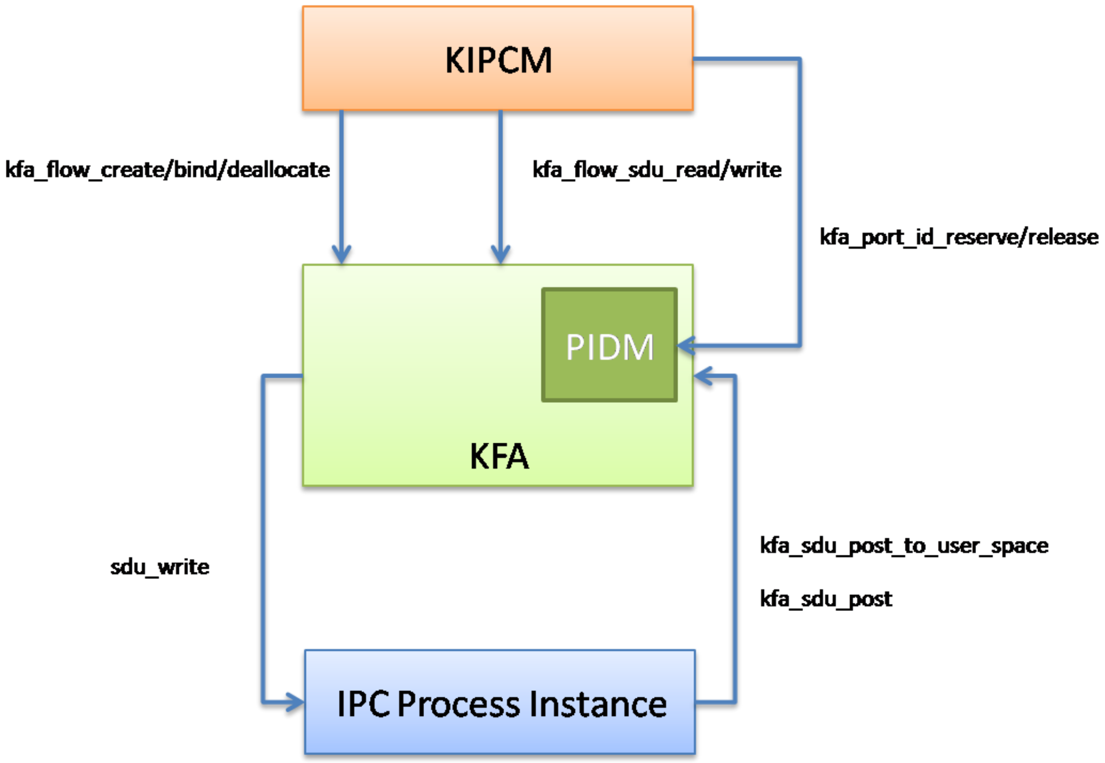
KIPCM
kfa_flow_create/bind/deallocate
kfa_flow_sdu_read/write
kfa_port_id_reserve/release
PIDM
KFA
sdu_write
kfa_sdu_post_to_user_space
kfa_sdu_post
IPC Process Instance

Figure 3 KFA Interactions (details)

2.1.4.2 KIPCM, the Kernel IPC Manager

As described in [3, section 7.5.1], the KIPCM is the counterpart in kernel-space of the IPC Manager in user-space. Its main responsibilities are the lifecycle management of several components in the kernel, such as IPC processes and the KFA, and providing the main interface towards user-space. The following figure shows the main interactions between the KIPCM and the other components in the system.

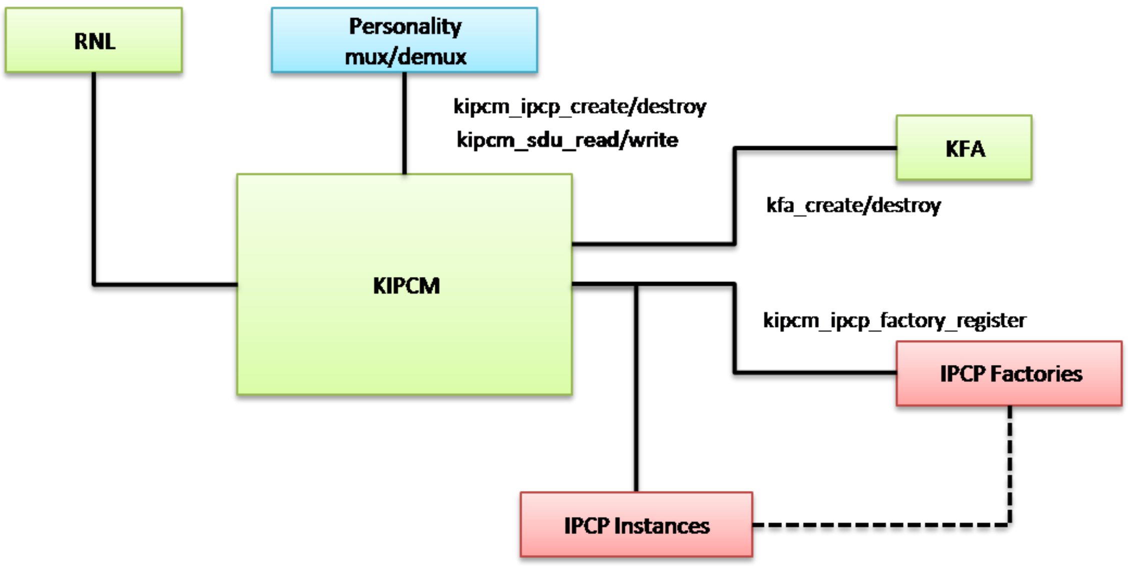

Figure 4 KIPCM interactions (details)

The OO approach introduced in section 2.1.1 shows its potential here: the KIPCM provides the same binding and usage APIs for both the normal and the shim IPC Processes, regardless of their “exact” type. This way, all the functionalities they provide are transparent (and homogeneous) to the upper layers, even though their inner workings may vary greatly. For instance, the “regular” IPC Processes have EFCP and RMT instances while the shims are completely missing them, but the user-space components are unaware of such differences. To provide such high-level abstraction, the KIPCM makes use of the IPC Processes factories. These factories are registered to the KIPCM after the loading of the corresponding IPC Process kernel module, adding to the system the capability to create the specific type of IPC Process with an independent interface.

Nevertheless, the specific functionalities offered by the KIPCM have been updated since [3], as already introduced in section **Errore. L'origine riferimento non è stata trovata.**, mostly moving flow management tasks to the KFA. They can be summarized as follows:

- It is in charge of the creation of IPC Processes in the kernel.
- It is in charge of retrieving the corresponding IPC Process instance when it has to perform any task triggered by a call to the to-user-space interface (i.e. flow allocation, register application to IPC process, etc).
- It abstracts the nature of the IPC Process to the applications by providing a unique API for reading and writing SDUs.
- It is the main hub to the RNL Layer presented in section 2.1.4.3, correlating incoming/outgoing (replies/requests) messages by means of their sequence numbers.

2.1.4.3 RNL, the RINA Netlink Layer

In conjunction with a set of system calls and procfs/sysfs, Netlink sockets are one of the three technologies that shape the communication framework between user space processes and the kernel in the IRATI prototype.

Netlink is a flexible, socket-based, communications channel typically used for kernel / user-space dialogues. It defines a limited set of families, each one dealing with a specific service. The family selects the kernel module - or Netlink group - to communicate with, e.g. NETLINK_ROUTE (receives routing and link updates and may be used to modify the routing tables), NETLINK_FIREWALL (transports IPv4 packets from the netfilter subsystem to user space) etc.

In order to counteract this limited amount of available families, the Generic Netlink extension was introduced as another Netlink family that works as a protocol type multiplexer, increasing the number of families supported by the original Netlink specification. Please refer [3, section 7.3.1.3] and [4] for further details.

The RINA Netlink Layer is the solution implemented in the IRATI stack to integrate Netlink in the RINA software framework. It presents an abstraction layer that deals with all the tasks related to the configuration, generation and destruction of Netlink sockets and messages, hiding their complexity to the RNL users.

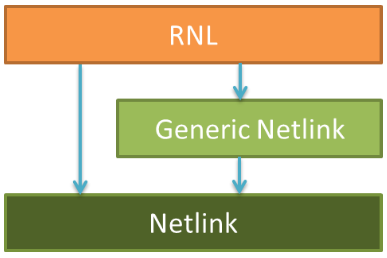

Figure 5 RNL and Netlink layers in the IRATI prototype

On the one hand, RNL defines a Generic Netlink family called NETLINK_RINA for the exclusive use of the RINA software framework. This family describes the set of messages that can be sent or received by the components in the kernel. On the other hand, the RNL acts as a multi-directional communication hub that multiplexes the messages upon reception from user space and provides message generation facilities to the different components in the kernel.

Upon initialization, the KIPCM registers a set of handlers towards the RNL, one for each message defined in the stack's RNL API. The RNL handlers are objects that contain a call-back function and a parameter that is opaquely transferred by RNL to the receptor of the call-back once it gets invoked. This approach allows each component of the kernel to register a specific handler with a call-back function that is in charge of performing the task associated to the type of the message received.

The RNL behaviours on ingress and egress directions can be summarised as follows:

- Upon reception of a new message: the RNL looks in the registered set for the proper handler and forwards the message by invoking the registered call-back function, passing the responsibility of reacting to the message to the component in charge. This component relies on the second set of utilities provided by the RNL: parsing and formatting of the NETLINK_RINA family messages. Upon reception, the component will call the RNL in order to parse the received message. RNL will validate the message correctness (i.e. the message is well formed and its type and parameters are correct). It will return a well-known internal structure with the translation of the parameters contained in the original message.
- In the output direction, when a component in the kernel sends a Netlink message to user-space: the component will call RNL passing the message type and the value of the parameters to be sent. The RNL will format the corresponding Netlink message attaching a sequence number to it. Finally, via a second call to RNL, it will send the resulting message to its destination (identified by the Netlink socket number).

Figure 6 depicts the most important interactions between the RNL and the KIPCM. Each number represents an ordered step in the process: 1) At bootsrap, the KIPCM registers a RNL set; 2) The KIPCM registers the handlers required for this registered RNL set; 3) when a message of certain type is received from user-space, the corresponding handler is retrieved and the callback to the module in charge is called, triggering the task to be performed ; 4) If a response message is needed it is sent using RNL again.

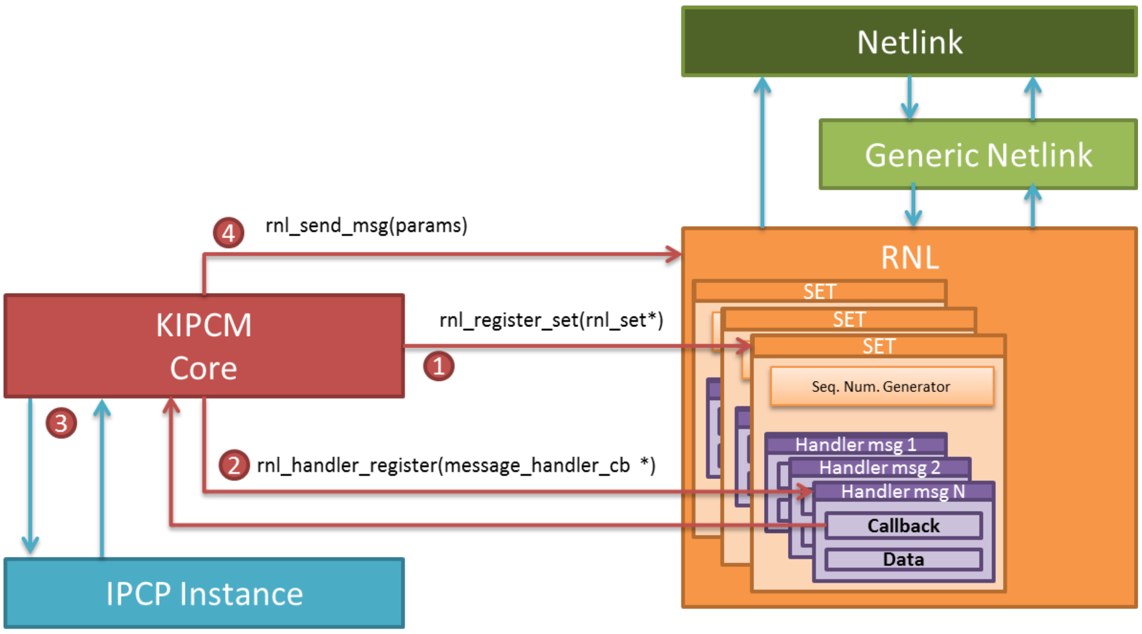

Figure 6 RNL interaction (detailed)

2.1.4.4 The IPC Process factories

The IPC Process factory concept has been introduced into the IRATI kernel-space stack in order to abstract the real nature of the IPC processes – normal or shim – to their users (e.g. KIPCM, the user-space). It is an abstract constructor of IPC processes, as its OOD factory concept counterpart.

Each IPC Process (e.g. the Shim IPC Process over Ethernet, the Normal IPC Process) has its ad-hoc factory that registers to the KIPCM during its initialization (e.g. when the IPC Process module is loaded). The KIPCM, in turn, uses the provided factory whenever an IPC Process instance of the corresponding type has to be created or destroyed.

All the factories in the system provide instances of IPC processes with the same common interface (i.e. with the same “methods” signatures). Nevertheless, each IPC Process instance created is underneath bound to its specific type. This is possible because, although the factory provides a common API for IPC Process creation/destruction to the core components, it knows the particularities of the specific type of IPC process it constructs.

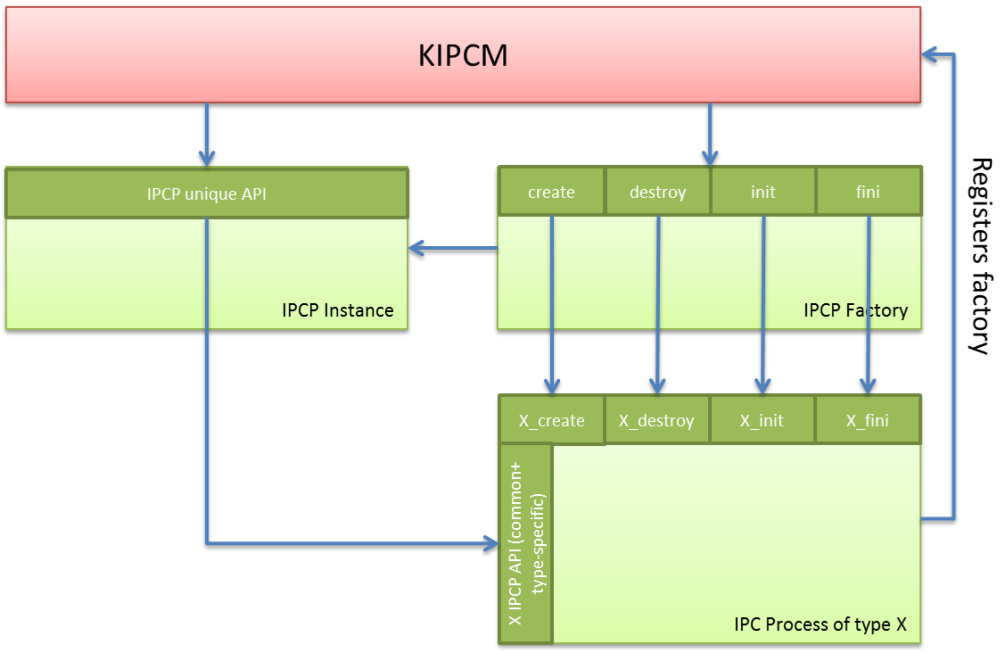

Figure 7 IPCP Factory, IPCP instance and IPCP interactions

2.1.4.4.1 The IPC Process factories interface

The IPC Process factories interface follows the rules for any object interface in the IRATI framework (ref. section 2.1.1.2). It is made up of four methods, two addressed to the initialization/finalization of the factory and two for the creation/destruction of IPC processes:

- **init:** Initializes the factory's internal structures (e.g. at module loading time).
- **fini:** Destroys any factory's internal structures and frees allocated memory (e.g. at module unloading time).
- **create:** This call is invoked by the KIPCM so the factories create an abstract IPC Process instance whose API is bound to the particular API of an IPC process of the factory specific type (ref. section 2.1.5.1).
- **destroy:** This call is invoked by the KIPCM to destroy an IPC Process instance.

2.1.5 The IPC Processes

In deliverable D2.1 [3], the necessity to distribute the different components of the normal IPC Process between user and kernel spaces was identified. This distribution minimizes the number of context-switches during the fast-path related operations. The partitioning brought the EFCP, RMT, PDU Forwarding Table and SDU protection components to kernel space as part of a normal IPC Process instance. However, for the shim IPC Processes none of their components can lie outside the kernel, since the internal components are different and specific for the underlying technology (i.e. Ethernet).

Nevertheless, the interface and the internal structures of normal IPC Processes have been slightly updated from what is stated in [3]. The following sections briefly describe such updates and introduce the inner workings of the IPC Processes in the prototype.

2.1.5.1 The IPC Processes interfaces

Shim and normal IPC Processes offer both a common and a type-dependent interface. The common interface allows the seamless binding of instances to other kernel components, regardless of its type. It consists of a truly common part, used to assign IPC Processes to DIFs, register applications, and read and write from and to flows. The specific part of the interface addresses the type-related functionalities of the IPC process. These particularities make reference to the way flows are created and managed in normal and shim IPC Processes and the differences between the internal structures in each one and their configuration requirements. The aggregation of both common and specific set of calls forms the unique API that is provided by the IPC Process instance object.

2.1.5.1.1 The normal IPC Process interface

As already explained in the previous section, the normal IPC Process API comprises a subset of specific calls depending on the type of the IPC Process (normal in this case); and a subset of calls that are common to any IPC Process.

The common set of calls is:

- **assign_to_dif**: This operation is triggered by the IPC Manager in user-space. The affected IPC Process receives all the necessary information on the DIF in order to be able to start operating as part of it.
- **update_dif_config**: This call passes a new configuration (after change) of an associated DIF to the IPC Process.
- **sdu_write**: When an application has been granted a flow towards another application and wants to send a SDU, it makes use of the available system call present in the user-space/kernel interface. This call is processed by the KIPCM which in turn calls the sdu_write function in the IPC Process that is supporting the flow.
- **flow_binding_ipcp:** This call binds the RMT structure of the (N) IPC Process to the (N-1) flow structure from/to which the (N) IPC Process will receive/send SDUs.

The specific set of calls is:

- **connection_create:** As part of the flow allocation process for a normal IPC Process, at least a connection must be created to support the requested flow. This call creates the EFCP instances required in the kernel part of the normal IPC Process.
- **connection_destroy:** For the opposite situation, this call destroys an EFCP instance.
- **connection_update:** During the connection set up process, this call is used to update an EFCP instance with the connection identifier of the peering EFCP instance (the EFCP instance at the opposite end of the connection).
- **connection_create_arrived:** During the connection set up process, this call is used by the KIPCM to notify a normal IPC process that another normal IPC Process at the opposite end is requesting a connection to support a flow.
- **management_sdu_write:** Invoked by the IPC Process Daemon at user-space, when it wants to send a layer management SDU to a peer IPC Process through a given port-id. The kernel parts of the IPC Process will add the required DTP header to the SDU, and schedule the resulting PDU for transmission through the Relaying and Multiplexing Task (RMT).
- **management_sdu_read:** Invoked by the IPC Process Daemon to retrieve the next layer management SDU directed to it. When the kernel components of the IPC Process detect that an incoming PDU contains a layer management SDU, they store the payload of the PDU – as well as the port-id it arrived from – in an internal queue. When the IPC Process Daemon invokes the management_sdu_read system call, the IPC Process kernel components return the first element of the queue or make the calling process sleep until there is an element available.
- **pdu_forwarding_table_modify:** Invoked by the PDU Forwarding Table computation entity at the IPC Process Daemon, in order to add or remove entries of the PDU Forwarding Table.

2.1.5.1.2 The shim IPC Process interfaces

As already stated, the common part of the shim IPC Process interface is the same as explained in previous section. The specific set of calls offered by this type of IPC Process is:

- **application_register**: As result of this operation triggered by an application via the IPC Manager in user-space, that application is registered as reachable from the IPC process. The IPC Process may spread this information to the rest of the DIF members.
- **application_unregister**: Complementary to **application_register**.
- **flow_allocate_request:** This call is invoked by the KIPCM when an application process on top requests a new flow to a destination application via the IPC Manager in user-space.
- **flow_allocate_response:** Informs the IPC Process about the application’s decision with regards to accepting or denying the flow allocation request. The IPC Process will react committing the flow in the former case, or deleting it in the latter.

- **flow_deallocate:** Deallocates an existing flow, freeing the corresponding resources.

2.1.5.2 EFCP, RMT and the PDU Forwarding Table (in the normal IPC Process)

The EFCP instance implements the Error and Flow Control Protocol state machine. It holds the DTP and (optionally) DTCP instances associated to the flow. For the 1st prototype, only the DTP state machine has been implemented in order to support the flows among different IPC Processes in the same system while for the DTCP only basic placeholders have been allocated. The basic functionalities of EFCP are supported and it provides its services through the APIs summarized as follows:

- **efcp_write:** Injects a SDU into the instance, to be processed and delivered to the RMT (outgoing direction).
- **efcp_receive:** Takes a PDU as an input, processes the DTP PCI and delivers any resulting complete SDUs to the N+1 IPC Process or to the queues of SDUs ready to be consumed by user-space (incoming direction).

Although a flow may be supported by several connections, in the current prototype there is one active EFCP instance per flow. Nevertheless, the EFCP Container concept has been introduced to abstract the notion of multiple EFCP instances in the same IPC Process. The EFCP Container provides the efcp_container_write and efcp_container_receive APIs. These APIs mux/demuxes incoming requests onto the particular EFCP instance that will finally perform its task. The EFCP Container holds ingress and egress workqueues that are used to sustain the work of each EFCP Protocol Machine into the container. These workqueues bind data (e.g. the SDU) to code (e.g. the function processing the SDU) deferring execution from API calls. Therefore, they are the means to decouple the ingress and egress workflows within the EFCP component. Finally, the EFCP Container manages the CEP identifiers (the Connection End Point identifiers of flows, unique within the IPC Process instance scope).

Conversely, there is only one RMT instance per IPC Process that, similarly to the EFCP Container counterpart, holds ingress and egress work queues for the purposes formerly presented (i.e. decouple and sustain the DU workflows into RMT).

The EFCP Container and the RMT instance coupled together sustain the ingress and egress DU workflows within a normal IPC Process. Figure 8 summarizes the interactions of these components with the rest of the stack during the “write” (green) and “read” (red) operations.

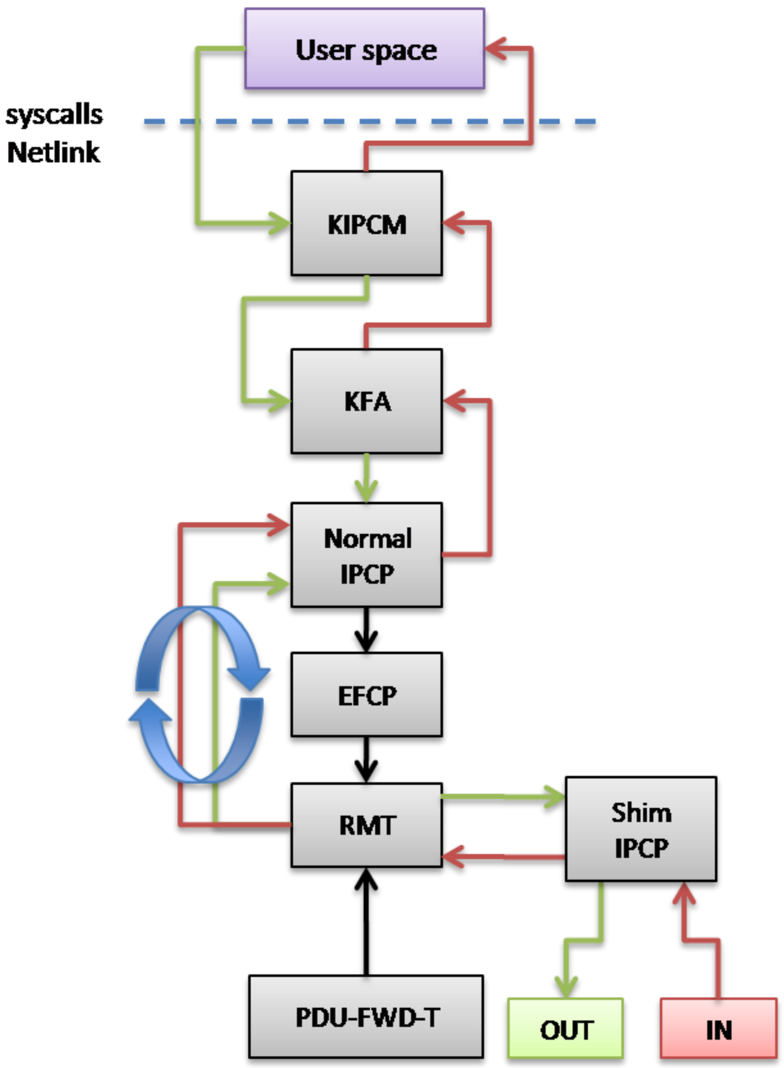

Figure 8 The components interactions during read/write operations

The following sections delve in the DUs workflows by exposing them at two different levels: at the *intra* IPC Process level, i.e. the part that takes place between the EFCP Container and the RMT, and at the *inter* IPC Process level, then the different IPC Processes in the IRATI stack communicate each other to serve a flow on the fast-path.

2.1.5.2.1 The egress DUs workflow

The intra IPC Process egress DU workflow can be summarized as follows:

1. A SDU arrives at the EFCP Container, via the efcp_container_write call.

2. The EFCP Container retrieves the associated EFCP instance and invokes the efcp_write call.
3. The EFCP instance posts the SDU into the DTP. This call creates a new workitem in the egress workqueue of the EFCP Container.
4. When the EFCP Container workitem is executed, the DTP is invoked:
    a. The DTP protocol machine creates a PCI and, with the SDU, builds a PDU.
    b. The PDU is sent to RMT by invoking rmt_send which, in turn, creates a workitem in its egress workqueue.
5. When the RMT workitem is executed, the RMT core:
    a. Retrieves the port-id that must be used to send the PDU given the destination address and the QoS id of this flow.
    b. Finally sends out the SDU crafted from the original PDU (this step is required because the N-1 DIF supporting the flow will treat the PDU as just a SDU).

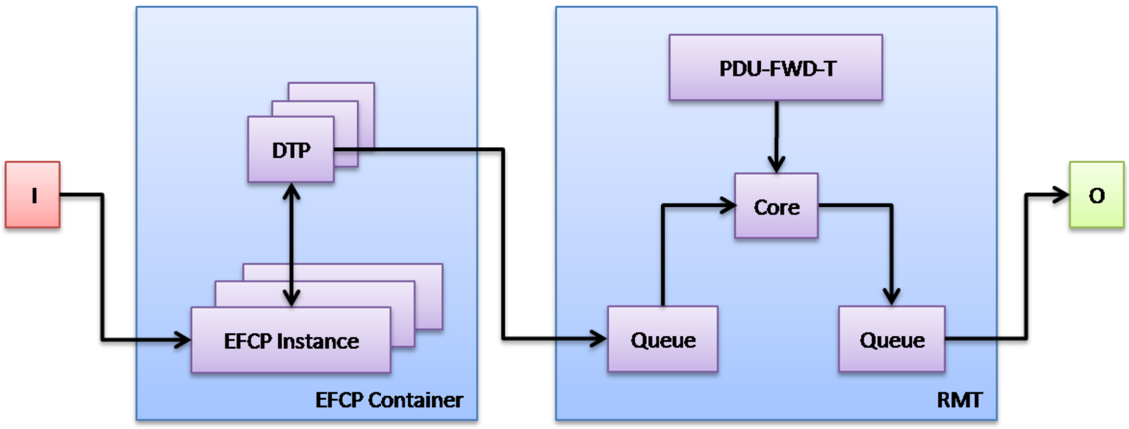

Figure 9 The egress EFCP / RMT workflow

Figure 10 shows the inter IPC Process workflow for an outgoing SDU through the DIF stack, i.e. the path a DU would follow through a stack of DIFs to exit the system towards the destination application.

In the example, 3 DIFs are stacked in a system, each one with its IPC Process representative. IPC Process 0 is a shim IPC Process, the representative of a shim DIF in the system. IPC Processes 1 and 2 are normal IPC Processes. The port identifiers in Figure 10 follow the convention of using the names or numbers of the underlying IPC Processes that are the origin and destination of this port id, e.g. “port id app2” means that it binds the application to the IPC Process 2. The workflow begins when an application, which has already been granted a flow to another application, invokes the syscall sys_sdu_write.

1. This syscall takes as input parameters the SDU to be sent and the port id of the flow. In this example, app2.
2. The KIPCM, via its kipcm_sdu_write API, receives the SDU and relays it to the KFA by invoking kfa_flow_sdu_write.
3. The KFA retrieves the IPC Process instance from the flow structure associated to this port id and invokes the sdu_write API in the IPC Process interface. In this case, IPC Process 2 provides normal_write function, which calls the efcp_container_write API.
4. The intra IPC Process path described before is followed from the call to efcp_container_write until the crafted SDU (sdu*) is sent to the KFA by means of kfa_flow_sdu_write. The port id retrieved by RMT 2 will be 21 (NOTE: the star in "sdu*" means that this SDU is different from the one originally received by the KIPCM).
5. The intra IPC Process path is followed again for IPC Process 1, which is normal as well. A new crafted SDU (NOTE: the double stars mean that it is different from the previous "sdu*") is sent to the KFA by RMT 1 with port id 10.
6. The KFA retrieves the flow structure and thus the IPC Process instance, invoking the sdu_write operation in the IPC process interface. However, this time the IPC Process is the shim IPC Process and therefore the sdu_write operation results in a call to shim_write. Finally, the shim follows its own scheme to send sdu** (ref. section 2.1.8).

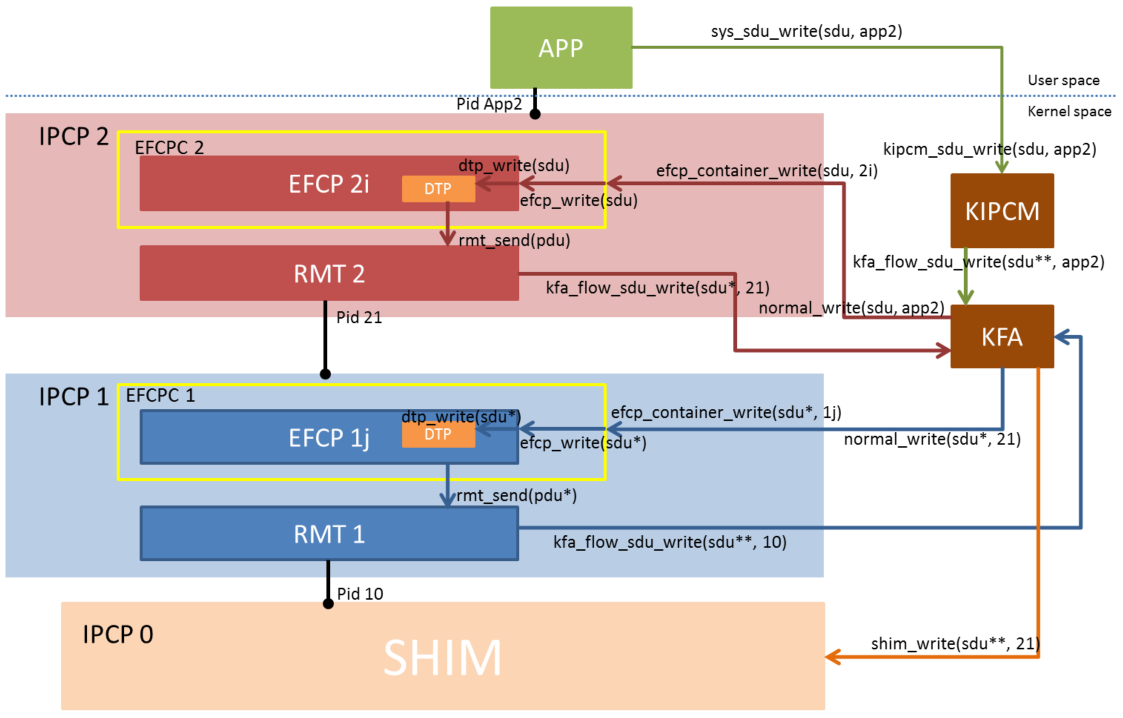

Figure 10 The egress DUs workflow

2.1.5.2.2 The ingress DUs workflow

The intra IPC Process ingress DU workflow can be summarized as follows:

1. A SDU arrives in the RMT by means of a call to rmt_receive and is transformed into an ingress workqueue (work)item.
2. When the workitem is executed, the RMT task will craft a PDU from the SDU and will check the destination address:
    a. If the destination address is not the address of the IPC Process it belongs to, it will look for a N-1 flow (using the PDU Forwarding Table) to send the SDU. In case there is not an entry, the SDU will be discarded.
    b. If the destination address is the address of the IPC Process, it will call efcp_container_receive to pass the PDU to the DTP. The task of processing the PDU will be inserted into the ingress workqueue.
        i. When the task is run:
            1. The DTP retrieves the connection information, i.e. the port-id, by means of the PCI (which will be discarded)
            2. Finally sends out the SDU by invoking the call to kfa_sdu_post.

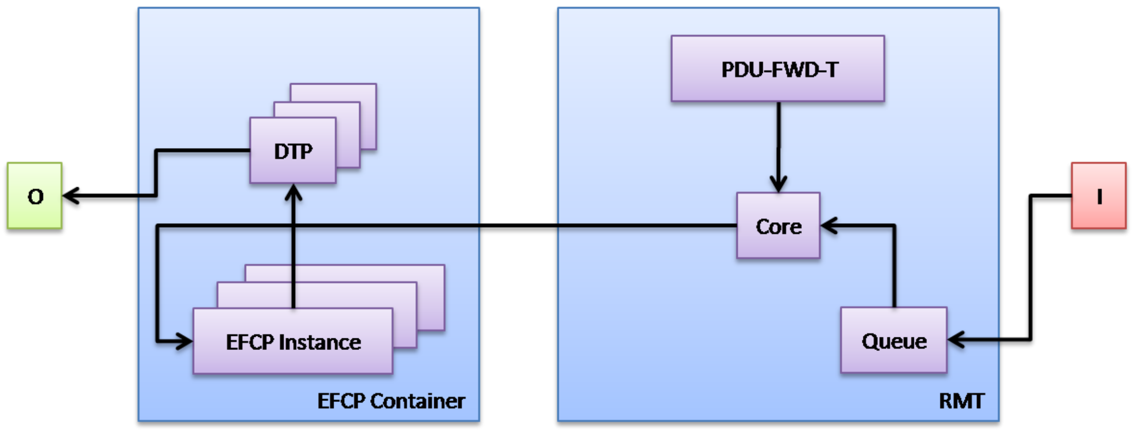

Figure 11 The ingress EFCP / RMT workflow

Figure 12 shows the inter IPC Process ingress DU workflow, i.e. an example of the path a DU would follow through a stack of DIFs to arrive at the destination application after entering the system. The setup is identical to the one described in section 2.1.5.2.1.

1. A SDU arrives in the system and after being processed by the device layer is received by the shim IPC Process (IPCP 0 in the figure). IPCP 0 performs the necessary steps (ref. section 2.1.8) to identify the port to which the SDU has to be forwarded - pid 10 in the figure - and it calls the kfa_sdu_post operation to send the SDU (sdu**).
2. The KFA retrieves its flow structure and looks for a RMT instance associated to this flow (ref. section 2.1.5.1.1). It uses the call rmt_receive to post the SDU (sdu**) on the RMT instance belonging to IPC Process 1.
3. The intra IPC Process path described before is followed from the call to rmt_receive until the call to kfa_sdu_post to finally send sdu* to the flow identified with port id 21.
4. Steps 2 and 3 are executed until the KFA does not find a corresponding RMT where to post the SDU to, considering the flow to be directed to an application in user space
5. The KFA pushes the SDU into the SDUs ready queue of the flow structure.
6. The application invokes the sdu_read syscall to retrieve the SDU (from the flow' ready-queue).

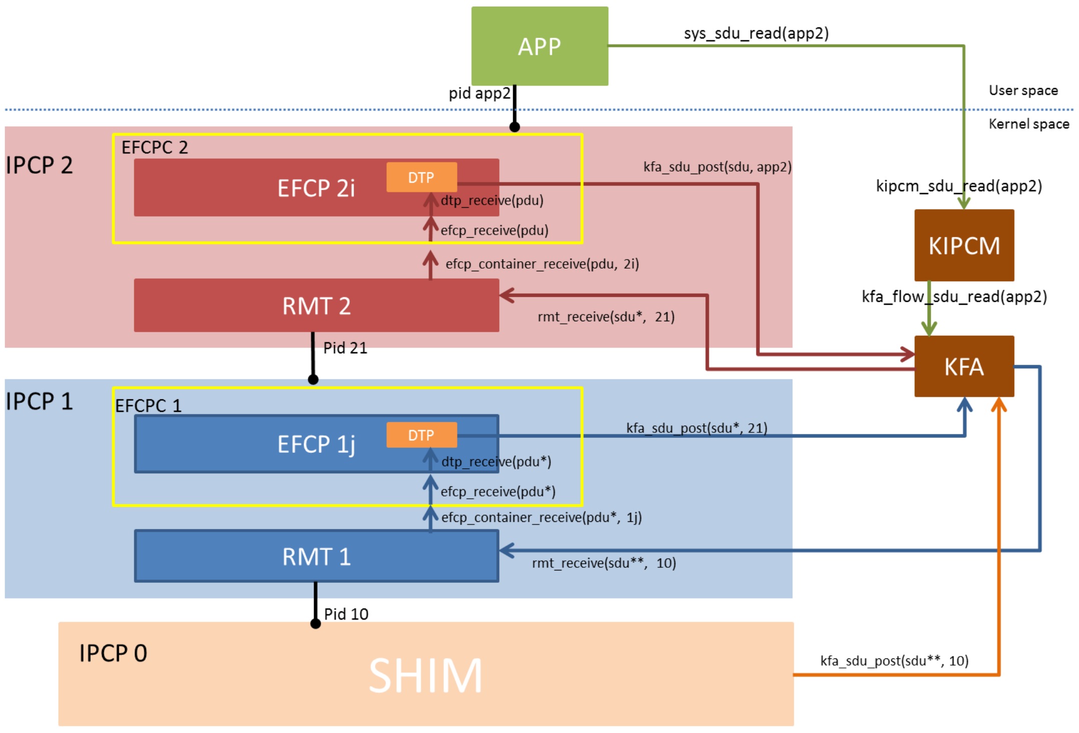

Figure 12 The ingress DUs workflow

2.1.6 ARP826

The ARP826 software component is a lightweight RFC-826 compliant ARP protocol implementation [26]. The main motivation for the development of this additional software component is the unfeasibility of the reuse of the current Linux ARP implementation (as in Linux kernel v3.10.0), mainly for the following reasons:

- It is too interwoven with the TCP/IP layer:
    - While the original RFC was designed to translate any network protocol address to an Ethernet address, the Linux ARP implementation only allows translating an IPv4 address to any hardware address. In that case, the Shim Ethernet IPC Process would have to enforce naming restrictions on the IPC processes using the shim. For instance the IPC Process names would have to be constrained to IP addresses.
    - When a new mapping entry becomes available in the ARP cache, only the TCP/IP stack is notified. Re-use of the Linux ARP implementation would either mean heavily patching the code to overcome this problem or actively polling the cache to see if a mapping becomes available.
- It only allows having one protocol network address per interface: Linux ARP gets the IP address from the device. This would require workarounds to assign the IPC process name to an interface in the form of an IP address. The TCP/IP stack would also have to use this IP address from then on.
- ARP requests can only be sent to IP addresses in the same subnet: This would make the naming restrictions even stricter, the shim IPC processes would be trickier to configure and the workarounds to circumvent this problem would render the solution hard to maintain and to propose mainstream.

With the IRATI ARP826 implementation, none of aforementioned restrictions hold. This implementation complies with RFC826, which assumes the generality the ARP protocol resolving any Layer 3 network address and therefore suits the Shim Ethernet IPC Processes requirements and, at the same time, avoids patching areas of the Linux kernel (i.e. ARP, IPv4, NetFilter) that would probably harm the adoption of the IRATI stack in the mainstream sources (i.e. the Linux kernel).

2.1.6.1 ARP826 software design

The ARP826 implementation is made up of different parts interacting with each other. These parts, depicted in Figure 13, can be summarized as: the *core*, *maps* and *tables* (also referred to as the "cache"), the *Address Resolution Module* (ARM) and the *RX/TX* part. All the parts together implement the ARP826 component as a dynamically loadable module (i.e. *arp826.ko*). The following sections briefly describe the ARP826 addresses abstraction, its parts and their interactions.

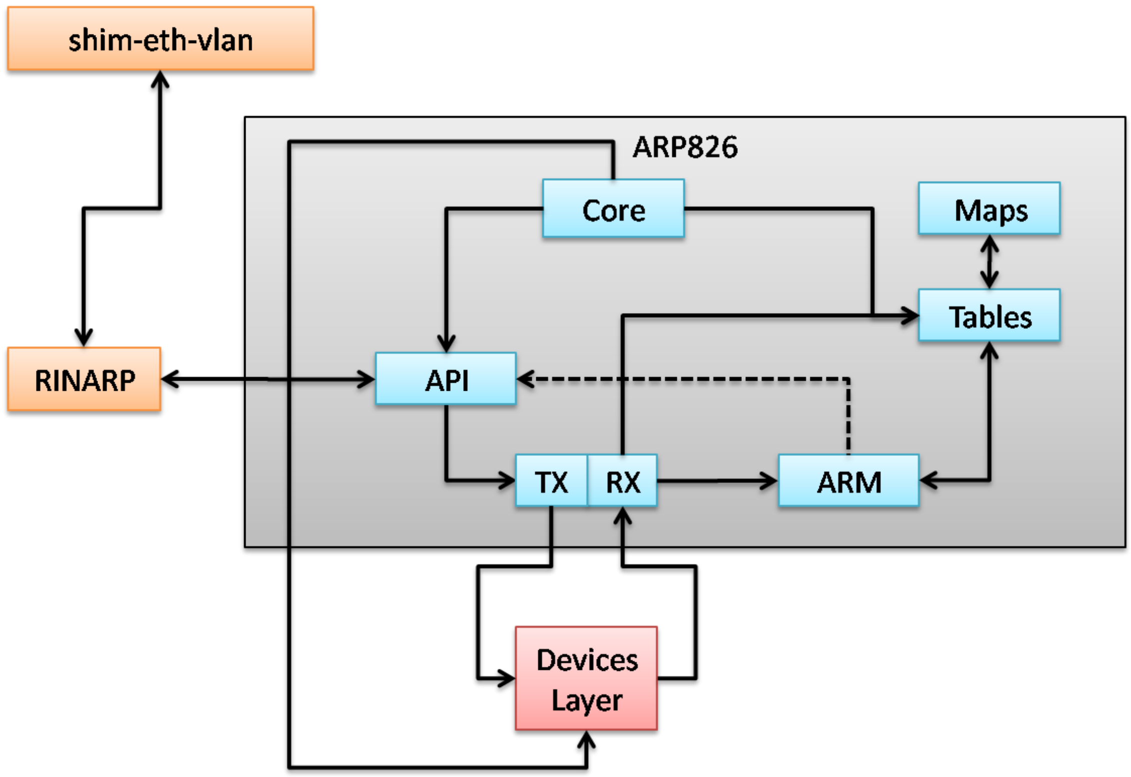

Figure 13 Interactions of the ARP826 parts

2.1.6.1.1 The Generic Protocol Addresses and Generic Hardware Addresses

The ARP826 component can be summarised as a directory that maps a protocol address to a hardware address. The data structures that symbolize these addresses are called Generic Protocol Addresses (GPAs) and Generic Hardware Addresses (GHAs). Their representation in the stack can be summarized as follows:

```
struct gpa {
      uint8_t * address;
      size_t    length;
};

typedef enum {
      MAC_ADDR_802_3,
      ...
} gha_type_t;

struct gha {
      gha_type_t type;
      union {
            uint8_t mac_802_3[6];
            ...
      } data;
};
```

These representations are hidden for the user, following the kernel-space object-oriented approach described in section 2.1.1, while functions to handle such objects are available as part of their API.

The GHA/GPA abstraction provides many services to its users, most importantly:

- Addresses growing/shrinking: In an ARP packet there is a field that specifies the length of the network protocol addresses. In some network protocols the length of the network address can vary. Therefore, there are also methods to grow and shrink GPAs, by adding and removing filler data. This way the length is set on a per packet basis.
- Per-technology GHAs creation: The GHA component allows the creation of special GHAs, such as the "broadcast" and "unknown" GHAs (i.e. the address that must be inserted into ARP requests) for specific technology

2.1.6.1.2 The ARP826 Core

The core is in charge of the ARP826 module initialization and the registration of new network protocols in the ARP826 component.

Upon the component initialization, the `arp_receive` function implemented in the ARP826 RX/TX part, is registered to the devices layer frame-reception event (system-wide). This function is invoked to handle incoming Ethernet frames with ARP ethertype. It handles ARP packets for registered network protocols.

2.1.6.1.3 The ARP826 Maps and Tables

ARP826 Maps and ARP826 Tables are the data structures supporting the ARP826 implementation:

The ARP826 Tables are the main abstraction supporting addresses mapping, commonly referred to as the ARP cache in other implementations. ARP826 Tables are composed by Table-entries, maintaining mapping between GPAs and GHAs. Table-entries can be freely added, removed, updated by network address, retrieved by hardware (GHA) / network address (GPA) / both. Tables lookup and management operations (e.g. insertion and removal of table entries) are mainly performed by the RX/TX and ARM components.

The ARP826 Maps are the internal data structure used by the ARP826, effectively storing the mappings as hash-maps.

2.1.6.1.4 The ARP826 Address Resolution Module

The ARP826 Address Resolution Module (ARM) is in charge of handling the addresses resolution requests. An application that is using the ARP826 component calls the ARP request API call when it wants to obtain a mapping of a network address (GPA) to a hardware address (GHA). The API call contains the source and destination network and hardware addresses, as well as a call-back that will be invoked when the mapping becomes available. The API can be summarized as follows

```
typedef void (* arp826_notify_t)(void *                opaque,
                                 const struct gpa * tpa,
                                 const struct gha * tha);
int arp826_resolve_gpa(struct net_device * dev,
                       uint16_t            ptype,
                       const struct gpa *  spa,
                       const struct gha *  sha,
                       const struct gpa *  tpa,
                       arp826_notify_t     notify,
                       void *              opaque);
```

The combination of all the `arp826_resolve_gpa` parameters (i.e. `dev`, `ptype`, `spa`, `sha`, `tpa`, `notify` and `opaque`) is added to a list of on-going resolutions when the function is called. The list allow the decoupling of the applications request (an ongoing resolution) from the ARP Protocol Machine.

Upon receipt of an ARP reply, a new workitem is created and added to the workqueue of the ARM component. A workqueue is used here to decouple the ARM frame handler from the non-interruptible context the reception of new frame poses.

When the workitem is finally executed, the list of on-going resolutions is browsed through to look for an entry matching the request (`dev`, `ptype`, `spa`, `sha`, `tpa` are used for the match). In the case of a match, the `notify` handler function is called and the entry gets removed from the ongoing resolution list. The request gets discarded otherwise.

2.1.6.1.5 The ARP826 RX/TX

The ARP826 RX/TX component is the very bottom part of the IRATI Stack ARP implementation. It can be viewed as a passive component in the sense that it only responds to external events:

- The receiving (RX) part responds to an incoming new frame. It checks if it can process it (i.e. it performs preliminary checks) and admits only ARP request or reply frames. If the frame is an ARP request, the ARP cache is checked to verify that the requested network address was registered on this system and, if this is the case, an ARP reply is sent back. If the frame is an ARP reply, it is handed to the ARP826 Address Resolution Module
- The transmitting (TX) part is executed when the API is called, and simply creates a new ARP request and transmits it on the specified device

2.1.7 RINARP

The RINARP component is an abstraction layer interposed between the Shim Ethernet IPC Process and the ARP826 components. RINARP decouples the Shim Ethernet IPC implementation from ARP826, thus allowing parallel developments of the two components.

The RINARP component is expected to be reused by other Ethernet based shims (e.g. a Shim-WIFI IPC Process). With the RINARP/ARP826 approach, the current ARP826 can evolve, even with major API changes, without involving significant updates to the shims already available (i.e. the Shim Ethernet IPC Process).

2.1.7.1 The RINARP API

The RINARP API is defined as follows:

```
struct rinarp_handle;

typedef void (* rinarp_notification_t)(void *             opaque,
                                       const struct gpa * tpa,
                                       const struct gha * tha);

struct rinarp_handle * rinarp_add(struct net_device * dev,
                                  const struct gpa *  pa,
                                  const struct gha *  ha);

int              rinarp_remove(struct rinarp_handle * handle);

int              rinarp_resolve_gpa(struct rinarp_handle * handle,
                                    const struct gpa *      tpa,
                                    rinarp_notification_t  notify,
                                    void *                  opaque);

const struct gpa *rinarp_find_gpa(struct rinarp_handle * handle,
                                  const struct gha *      tha);
```

The RINARP user must first get a handle through a call to `rinarp_add()`, specifying its (PA, HA) mapping. This handle will be used for any later call to the RINARP API. When the user finishes using RINARP, the handle must be disposed calling the `rinarp_remove()` function.

The API allows for asynchronous resolutions and synchronous lookups:

- To resolve a GPA asynchronously, `rinarp_resolve_gpa` has to be used. The function initiates the task of issuing the ARP request (using the target network protocol address, `tpa`). Once the resolution becomes available, the user-provided call-back function (i.e. `notify`) is invoked to notify about the resolution results.
- Since the Ethernet II standard has no flow allocation mechanism, the shim IPC process over Ethernet creates a new flow upon the reception of a new Ethernet frame [3]. However, the Ethernet frame only contains the hardware address, which means the sending application might be unknown. To solve this problem, a explicit ARP826 table lookup functionality has been added and the `rinarp_find_gpa` introduced. `rinarp_find_gpa` allows the reverse lookup operation (i.e. gets the GPA from the ARP cache, based on the corresponding GHA), supplying to the IRATI stack the mean to get the information on the flow initiator it requires.

2.1.8 The Shim IPC Process over Ethernet

The Shim IPC process over Ethernet, as specified in [3], has been implemented in the stack. It can be used by normal IPC processes or for testing purposes by applications, such as the echo test application. Since it must implement the IPC process factories interface as described in section 2.1.4.4, it provides the following operations:

- `eth_vlan_create`: Creates a new instance of the Shim IPC process over Ethernet and allocates the necessary memory for its data structures.
- `eth_vlan_destroy`: Cleans up all memory and destroys the instance.
- `eth_vlan_init`: Initializes all data structures in the instance.
- `eth_vlan_fini`: Cleans up all the memory and destroys the factory data.

The IPC Process instance this shim returns upon `eth_vlan_create` strictly follows the rules described in section 2.1.5.1. However, a freshly created and initialized instance is not yet functional since It first has to be assigned to a DIF. `eth_vlan_assign_to_dif` supplies the Shim IPC process over Ethernet with the information it needs to be able to start operating properly (i.e. the interface name and the VLAN id, which is the DIF name). Finally, a packet handler is added to the device layer (`eth_vlan_rcv`).

An application or IPC process that uses this Shim IPC process can consequently call the following operations, to start the inter processes communication:

- `eth_vlan_application_register`: Registers an application in the shim DIF, and thus in the ARP cache. Only one application can use the shim IPC process at a time. Because ARP does not differentiate between a client and a server application, every application has to call this operation before calling any other flow-related operation.
- `eth_vlan_application_unregister`: Unregisters the application in the shim DIF and removes its traces from the ARP cache. A different application can now use the shim IPC process.
- `eth_vlan_flow_allocate_request`: When this operation is called, a new flow is created and an ARP request is sent out, as specified in [3]. The `rinarp_resolve_handler function` is handed to the ARP826 component as the function that should be called upon receipt of the corresponding ARP reply.
- `eth_vlan_flow_allocate_response`: When the destination application answers about a flow allocation request, the shim is notified. This function is in charge of handling the application response as well as to update the flow status accordingly.
- `eth_vlan_flow_deallocate`: Deallocates the flow; cleans up all data structures related to the flow.

- `eth_vlan_sdu_write`: When a flow is allocated, SDUs can be written. This function encapsulates the SDU into an Ethernet frame, and sends it out of the device.

The Shim IPC process over Ethernet also has internal operations, which are needed for a working Shim IPC process over Ethernet implementation:

- `rinarp_resolve_handler`: Is called upon receipt of an ARP reply to an ARP request that was sent out when `eth_vlan_flow_allocate_request` was called. The flow is then allocated, and ”write” operations are finally allowed.
- `eth_vlan_rcv`: Receives new Ethernet frames, adds them to a list and queues a work item in the shim IPC process over Ethernet’s workqueue (in order to decouple from the non-interruptible context).
- `eth_vlan_rcv_worker`: Is called by the workqueue and processes at least one frame. For every frame in the list of frames `eth_vlan_recv_process_packet` is called.
- `eth_vlan_recv_process_packet`: Processes the packet with the following logic:
    - If the flow that corresponds with the sender’ MAC address in this frame is already allocated, it is delivered to the corresponding flow.
    - If the flow has not been allocated yet, but exists already, the frame is queued.
    - If the flow has been denied, the packet is dropped.
    - If the flow does not exist, a new flow is created and the KIPCM notified.

2.1.9 The Shim Dummy IPC Process

The Shim Dummy IPC Process can be imagined as a sort of loopback IPC Process. It is an IPC Process that does not traverse host boundaries and therefore cannot communicate with IPC Processes in other systems. It can be used by normal IPC processes or by applications, such as the echo test application, for testing purposes.

As any other IPC Process, the Shim Dummy registers its IPC Process factory into the system (KIPCM) upon initialization either during module loading (Shim Dummy built as a kernel module) or during kernel initialization (Shim Dummy embedded into the kernel). Since it must implement the IPC process factories interface as described in section 2.1.4.4, it provides the following operations:

- `dummy_create`: Creates a new instance of the Shim Dummy IPC process and allocates the necessary memory for its data structures.
- `dummy_destroy`: Cleans up all memory and destroys the instance.
- `dummy_init`: Initializes all data structures in the factory.
- `dummy_fini`: Cleans up all the memory and destroys the factory data.

The IPC Process instance the factory returns upon `dummy_create` strictly follows the rules described in section 2.1.5.1. However, a freshly created and initialized instance is not yet functional since it first has to be assigned to a DIF. The `dummy_assign_to_dif` "method" supplies the Shim Dummy IPC Process with the information it needs to start operating properly (the DIF name in this case).

The Shim Dummy IPC Process API that can be used by the applications or other IPC Processes, via the IPC Process instance abstraction, is the following:

- `dummy_application_register`: Registers an application in the Shim Dummy.
- `dummy_application_unregister`: Unregisters the application in the Shim Dummy.
- `dummy_flow_allocate_request`: When this operation is called, a new flow connecting two applications in the same system is created. The Shim Dummy notifies the destination applications and two flow structures are created in the KFA, one for the source application when the request is received by the KIPCM and the other for the destination application at this point.
- `dummy_flow_allocate_response`: When the destination application answers about a flow allocation request, the shim is notified. This function is in charge of handling the application response as well as to update the flow status accordingly.
- `dummy_flow_deallocate`: Deallocates the flow; cleans up all data structures related to the flow.
- `dummy_sdu_write`: Hook to the syscall sdu_write.
- `dummy_sdu_read`: Hook to the syscall sdu_read. A SDU from the sdu_ready queue is returned.

2.2 The user space

The user-space components are implemented into two different SW packages: librina and rinad.

The librina package contains all the IRATI stack libraries that have been introduced to abstract from the user all the kernel interactions (such as syscalls and Netlink details). Librina provides its functionalities to user-space RINA programs via scripting language extensions or statically/dynamically linkable libraries (i.e. for C/C++ programs). Librina is more a framework/middleware than a library: it has its own memory model (explicit, no garbage collection), its execution model is event-driven and it uses concurrency mechanics (its own threads) to do part of its work.

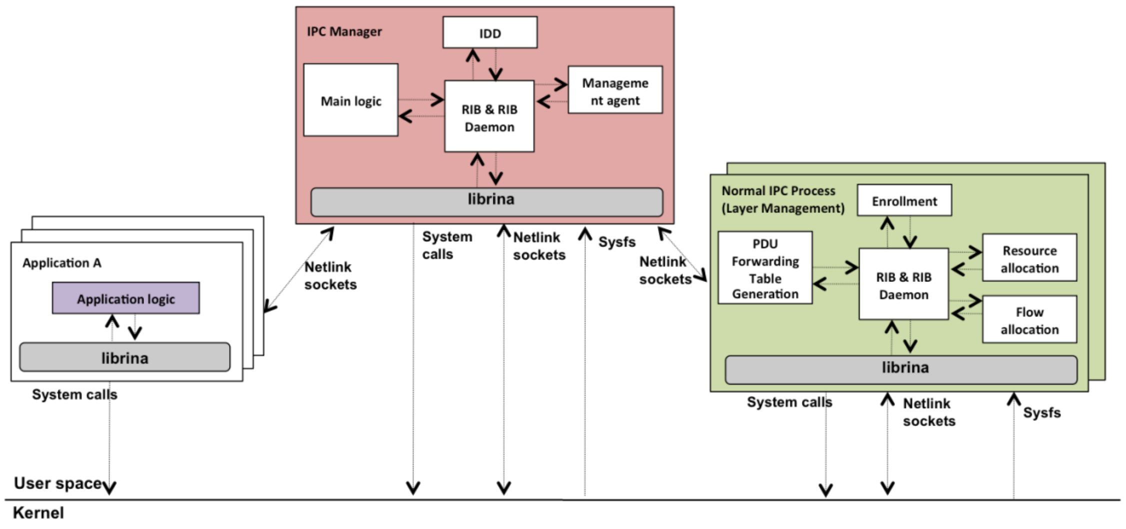

Figure 14 High-level architecture of the user-space parts

Rinad instead, contains the IPC Manager and IPC Process daemons as well as a testing application (RINABand). The IPC Manager is the core of IPC Management in the system, acting both as the manager of IPC Processes and a broker between applications and IPC Processes (enforcing access rights, mapping flow allocation or application registration requests to the right IPC Processes, etc.). IPC Process Daemons implement the layer management components of an IPC Process (enrollment, flow allocation, PDU Forwarding table generation or distributed resource allocation functions). For more details on the rationale behind this high-level architecture, interested readers might refer to the relevant sections in D2.1 [3].

Rinad also provides a couple of example/utility applications that serve two purposes: i) provide an example of how an application uses librina and ii) allow testing/experimentation with the IRATI stack by measuring some properties of the IPC service as perceived by the application (flow allocation time, goodput in terms of bytes read/write per second or mean delay).

In the following sections, the two software packages are described.

2.2.1 librina

The IRATI implementation provides the following libraries to support the operation of user-space daemons and applications.

- librina-application: Provides the APIs that allow an application to use RINA natively, enabling it to allocate and deallocate flows, read and write SDUs to that flows, and register/unregister to one or more DIFs.
- librina-ipc-manager: Provides the APIs that facilitate the IPC Manager to perform the tasks related to IPC Process creation, deletion and configuration.

- librina-ipc-process: APIs exposed by this library allow an IPC Process to configure the PDU forwarding table (through Netlink sockets), to create and delete EFCP instances (through Netlink sockets also), to request the allocation of kernel resources to support a flow (through system calls) and so on.
- librina-faux-sockets: Allow adapting a non-native RINA application (a traditional UNIX socket based application) to lay over the RINA stack.
- librina-cdap: Implementation of the CDAP protocol.
- librina-sdu-protection: APIs and implementation to use the SDU-protection module in user space to protect and unprotect SDUs (add CRCs, encryption, etc).
  librina-common: Common interfaces and data structures.

Librina enables applications and the different user-space components of the IRATI stack to communicate between them or with the kernel via Netlink sockets, system calls, or sysfs, hiding the complexity of dealing with these mechanisms. Librina also provides an object-oriented wrapper of the underlying threading facilities (e.g. libpthread), allowing librina users to take advantage of concurrency mechanisms without further external dependencies.

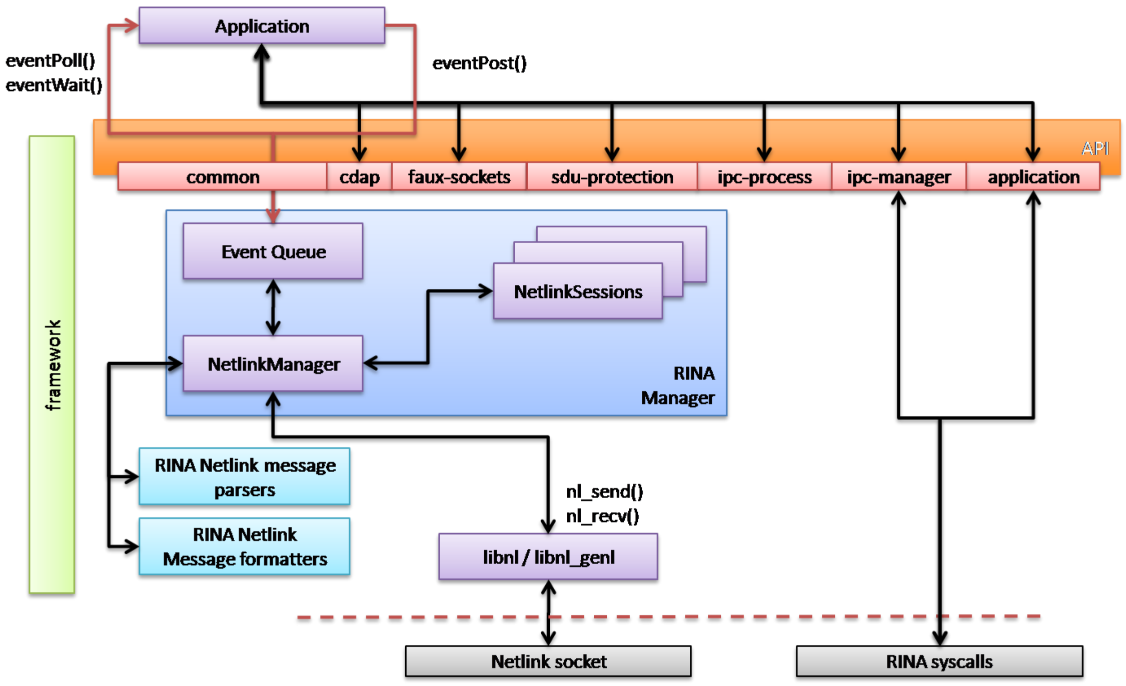

Figure 15 High-level librina design

Figure 15 shows the librina high-level design. Librina allows its users to invoke remote operations (handled by other OS processes) or system functions as if they were local calls to normal C++ objects. These proxy objects use librina internals to either send a Netlink message or to invoke a system call:

- Operations that result in system call invocations are synchronous. When the librina user invokes an operation of this type, the proxy class calls the librina internal handlers, which invoke the relevant system call and return the result to the proxy class. The proxy class, if required, formats the result and presents it to the caller.
- Operations that result in the generation of a Netlink message are asynchronous; the caller will get the result of the operation as an event. When the librina user invokes an operation of this type, the proxy class invokes the NetlinkManager, which provides an object oriented-wrapper to the libnl/libgnl libraries. The NetlinkManager generates a Netlink message, and uses libnl to send the message to the destination Netlink port-id. At this point the proxy class operation returns a handle to the caller so that he can identify the event reporting about the result of the operation (the handle is an integer).

Librina has an internal thread that continuously waits for incoming Netlink messages. When a message arrives the thread is woken up, requests the NetlinkManager to parse the message and adds the resulting event to an internal events queue. The librina user can get the available events by invoking the 'eventPoll' (non-blocking) or 'evenWait' operations in the API, which retrieve the element at the head of the events queue.

2.2.1.1 Bindings to other programming languages

The librina build framework has been enhanced in order to automatically generate bindings for interpreted languages through the use of SWIG [18].

SWIG is a software development tool that simplifies the task of interfacing different interpreted languages such as Perl, Python, Java and Ruby to C/C++ libraries. In simpler terms, SWIG is a compiler that takes C/C++ declarations and creates the wrappers required to access those declarations.

The wrappers SWIG generates are layered: the C/C++ declarations are bound to a C/C++ Low Level Wrapper (LLW) which in turn is connected, using the Native Interface (NI) semantics of the target language, to a High Level Wrapper (HLW). Both the LLW and HLW depend on the target language since they have to interact using different NIs (such as the JNI [23], the Python API [22] etc.).

Depending on the complexity of the library interface, SWIG has to be opportunely driven in order to produce good HL wrappers (and therefore target language modules or libraries suitable for the end-user). These corrections usually apply over an additional file (the SWIG interface file, also called the ".i" file), which can easily become an additional management burden.

Figure 16 shows a simple example of a C software module wrapping. In the example, the input software module (i.e. `example.c` and `example.h`) is exporting a function (i.e. the `fact`() function) and the relative SWIG driving directions for the binding productions are described in the interface file (i.e. the `example.i` file). Once the SWIG executable is executed with the interface file as input, it produces the low-level wrappers (i.e. `example_wrap.c`) and high-level wrappers (i.e `example.py`). Finally, The low-level wrappers are compiled as a dynamic library (i.e. `libexample.so`). This dynamic library will be loaded, on-demand, by the high-level wrappers once they are imported in the Python interpreter (i.e. `Python`).

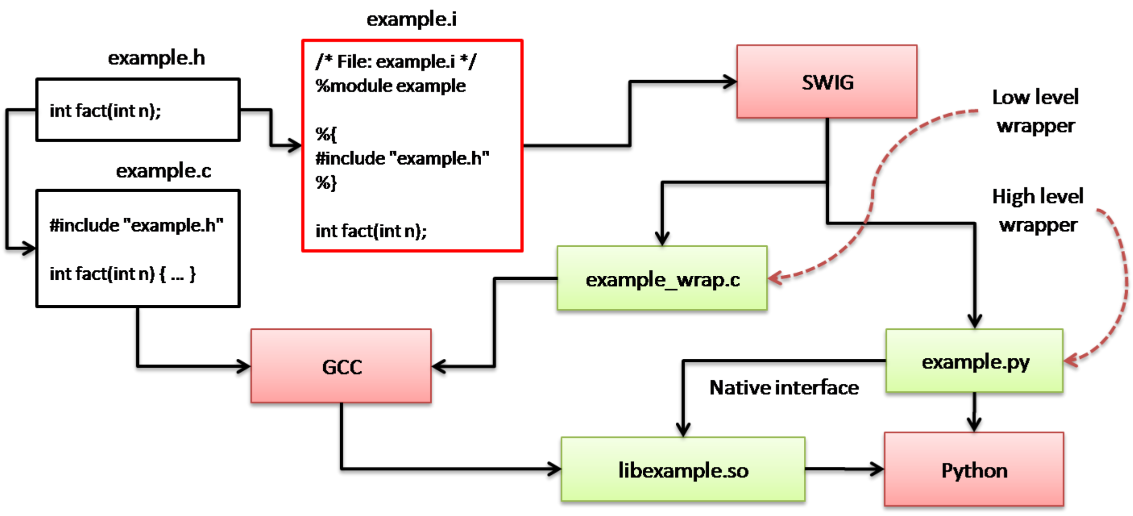

Figure 16 SWIG wrapping example

The wrapping of librina focused on minimising the efforts while reducing the maintenance costs of the bindings support to the bare minimum. To obtain the two goals a) the minimum corrections to the SWIG directions have been applied and b) the librina API stabilization has been prioritized over its internal mechanisms. As a matter of fact, the current Java SWIG interface files are ~500 Lines Of Code (LOCs) long which correspond ~13500 LOCs of automatically generated code (with a wrapper/code produced lines ratio of 1/27).

The approach has been embraced for a two-fold scope. It allows developing new applications in different programming languages, better suiting the needs the IRATI adopters may have, and it allows the reuse of existent software written in different languages, within the stack framework. The availability of the IPC Manager and IPC Process daemons – as well as the RINABand testing application – in the Alba Stack [20] were the primary drivers of such choice.

Bindings for other high-level interpreted languages such as Python are expected to be introduced in the future. Due to the reduced amount of wrapping directions for the Java parts and the better support SWIG has for other languages, their cost is expected to be far less than the cost of the Java bindings that have been introduced.

The following figure depicts the updated librina software architecture with respect to the architecture described in deliverable D2.1.

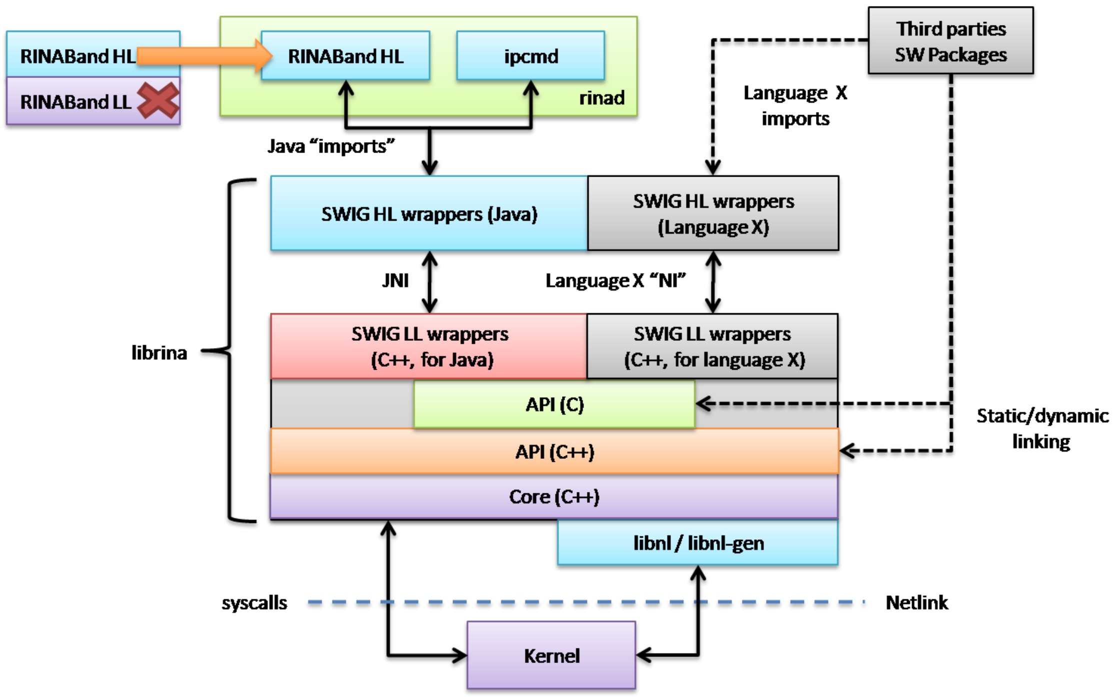

Figure 17 The IRATI user-space stack software architecture

2.2.1.2 Detailed Librina software design

Figure 18 provides a detailed overview of the internal components of librina, grouped in different categories. At the API level, user applications can find five types of classes:

- **Model classes**: These classes model objects that abstract different concepts related to the services provided by librina, such as: application names, flow specifications, RIB objects, neighbours and connections. Model classes contain information on the modelled objects, but do not provide operations to perform actions other than updating or reading the object's state.
- **Proxy classes**: These classes model 'active entities' within librina, meaning that they provide operations to perform actions on these entities. These actions result in the invocation of librina internals either to send a Netlink message to another user space

process or the kernel; or to invoke a system call. For instance, librina-application provides an 'IPCManager' proxy class that allows an application process to request the allocation or deallocation of flows to the IPC Manager Daemon. Another example can be found in the 'IPC Process' class available at librina-ipcmanager: this proxy class allows the IPC Manager daemon to invoke operations on the user-space or kernel components of an IPC Process.

- **Event classes**: As briefly introduced in section 2.2.1, librina is event-based. Invocation of proxy classes operations that cause the emission of a Netlink message return right away, without waiting for the Netlink message response. The response will be later obtained as one of the events received through the EventConsumer class. Event classes are the ones that encapsulate the information of the different events, discriminated by event type. Examples of events include results of flow allocation/deallocation operations or results of application registration/unregistration operations, just to name a few.
- **EventProducer**: This class allows librina users to access the events originated from the responses to the operations requested through the Proxy classes. The event producer provides blocking, non-blocking and time-bounded blocking operations to retrieve pending events.
- **Concurrency classes**: Concurrency classes provide an object-oriented wrapper to the OS threading functionalities. It is internally used by librina, but also exposed to librina users in case they want to use it as a way of avoiding external dependencies or intermixing different threading libraries (as it is the case of the IPC Manager and IPC Process daemons).

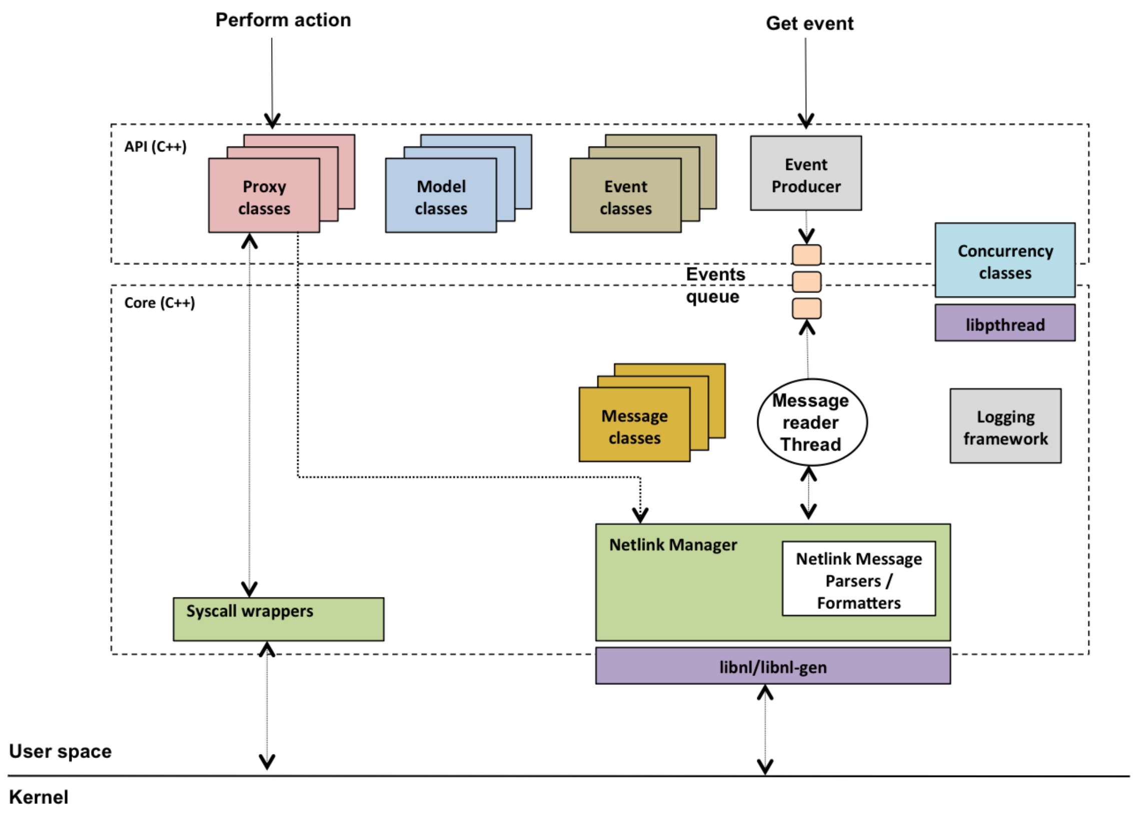

Figure 18 librina detailed software design

The librina core components process two types of inputs: operations invoked via Proxy classes at the API level or Netlink messages received via the Netlink socket bounded to librina - created at initialization time.

Operations invoked via proxy classes can follow two processing paths that either result in the invocation of a system call or on the generation of a Netlink message. In the former case processing is very simple: invocations of proxy operations are mapped to system call wrappers that make the required system call to the kernel (such as `readsdu`, `writesdu`, `createipcprocess` or `allocateportid`). The latter case involves more processing, as explained in the following:

- **Message classes**: These classes provide an object-oriented model of the different Netlink messages that can be sent or received by librina. The basic message class 'BaseNetlinkMessage' models all the information required to generate/parse the header of a Netlink message, including the Netlink header (source port-id, destination port-id and sequence number), the Generic Netlink family header (family and operation-code) and the RINA family header (source and destination IPC Process ids).

The different message classes extend the base class by modelling the information that is sent/received as Netlink message attributes in the different messages.

- **NetlinkManager**: This class provides an object-oriented wrapper of the functions available at the libnl/libgnl libraries (these libraries provide functions to generate, parse, send and receive Netlink messages). The wrapping is partial since only the functionality required by librina has been wrapped. In the 'output path' the NetlinkManager takes a message class, generates a buffer, adds the NL message header to the buffer, passes the message class and the buffer to the NL formatter classes (which will add NL attributes to the buffer) and finally passes the buffer to libnl to send the message. In the 'input path' – upon calling the blocking 'getMessage' operation – the IPC Manager blocks until libnl returns a buffer containing a NL message, then it parses the header, requests the NL parser classes to parse the NL attributes and return the appropriate message class, and returns.
- **NetlinkMessage Parsers/Formatters**: The goal of these classes is either to generate the attributes of a NL message based on the contents of a message class (formatting role) or to create and initialize a message class based on the attributes of a NL message (parsing role).

In order to ensure that all the NL messages are received in a timely fashion, librina-core has an internal thread that is continuously calling the blocking NetlinkManager 'getMessage' operation. When the operation returns the thread converts the resulting Message class to an Event class, and puts the Event class to an internal events queue. When a librina user calls the EventConsumer to retrieve an event, the EventConsumer tries to retrieve an element from the events queue by invoking the `eventPoll` (non-blocking), `eventWait` (blocking) or `eventTimedWait` (blocking but time-bounded) operation.

All librina components use an internal lightweight logging framework instead of an external one in order to minimize librina dependencies, since the goal is to facilitate deploying it within several OS/Linux systems.

2.2.2 rinad

Rinad contains the user-space daemons of the IRATI implementation, as well as two example applications that are also useful for testing the stack. Two types of daemons in user-space are responsible for implementing the IPC Manager and layer management IPC Process functionality.

- **IPC Process Daemon**: Implements the layer management parts of an IPC Process, which comprise the functions that are more complex but execute less often. These functions include: the RIB and the RIB Daemon, Enrolment, Flow Allocation, Resource Allocation and the PDU Forwarding Table Computation. There is one instance of the IPC Process Daemon per IPC Process in the system.

- **IPC Manager Daemon**: The central point of management in the system. It is responsible for the management and configuration of IPC Process daemons and kernel components; hosts the local management agent and IDD. There is one instance of the IPC Manager Daemon per system, but it can also be replicated in the future to increase availability.

The final goal of the IRATI prototype is to produce a C++ implementation of these daemons (for the 3rd prototype, end of 2014). This will allow maximizing the portability of the IRATI stack to different POSIX-based systems, as well as minimizing the code footprint and external dependencies. However, the aggressive timing of the 1st prototype, as well as the requirement of the first experimentation round to happen between months 7 and 11 of the project made the consortium take a different approach for D3.1: reusing part of the code of the Alba RINA stack [20] in order to implement the IRATI user-space daemons.

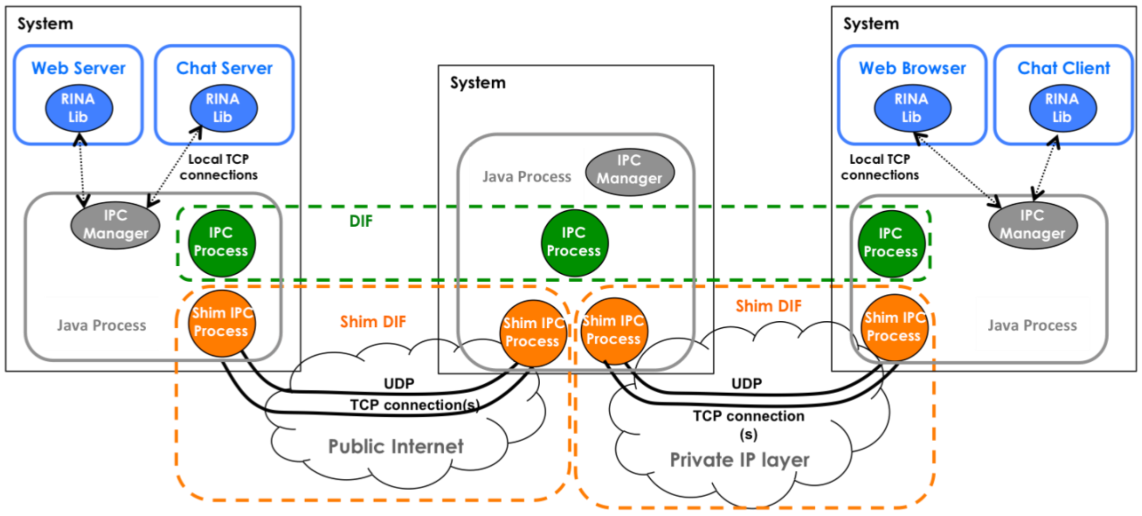

Figure 19 Alba RINA stack high level software design

Alba is a work-in-progress pure Java RINA prototype developed by i2CAT and the TSSG. All the RINA functionalities are implemented as a single OS processes running the Java Virtual Machine (JVM), as depicted in Figure 19. Java Applications can use an Alba-provided Java library to invoke the RINA functionality. This library communicates with the RINA OS process by means of local TCP connections. The Alba RINA prototype can only operate over TCP/UDP, since it is user-space only and therefore limited to what the OS sockets API can provide. Section 2.2.2.2 provides further detail on how the Alba code has been integrated into IRATI's phase 1 IPC Process and IPC Manager Daemons.

2.2.2.1 Package components and build framework

The librina features enhancement introduced in section 2.2.1.1, allowed the reuse of the Java code present in the Alba RINA stack. Such code, rearranged in order to use the librina bindings, has been introduced into the IRATI stack as the additional rinad package. This package provides:

- The IPC Process and the IPC Manager daemons, as described in D2.1.
- The RINABand testing application (for complex bandwidth-related tests).
- Echo client and server applications (for simple connectivity and configuration related tests).

The configuration and building framework templates, previously used in librina, have been updated to cope with the different rinad requirements. The configuration framework, initially planned only for C/C++ based code in librina, has been enhanced to look for JARs building related tools (such as the Java compiler). Maven (http://maven.apache.org) has been selected as the building framework for the Java parts. Therefore, the rinad package relies on the one hand on autotools for configuration and on the other hand on Maven for building.

2.2.2.2 RINA Daemons software design

The IPC Manager Daemon is the main responsible for managing the RINA stack in the system. It manages the IPC Process lifecycle, acts as the local management agent for the system and is the broker between applications and IPC Processes (filtering the IPC resources available to the different applications in the system). As introduced in section 2.2.2 the first phase prototype of the IPC Manager has been developed in Java, leveraging part of the Alba prototype codebases. Moreover, the current IPC Manager Daemon is not a complete implementation, since it does not implement the local management agent yet (therefore the RINA stack cannot be managed through a centralized DIF Management System).

Figure 20 shows a schema of the detailed IPC Manager Daemon software design. It is a Java OS process that leverages the operations provided by the librina API through the wrappers generated by SWIG and the Java Native Interface (JNI). In concrete, `librina-ipc-manager` provides the following proxy classes to the IPC Manager Daemon:

- **IPC Process Factory**. Enables the creation, destruction and enumeration of the different types of IPC Processes supported by the system.
- **IPC Process**. Allows the IPC Manager to request operations to IPC Processes such as assignment to DIFs, configuration updates, enrolment, registrations of applications or allocations/deallocations of flows.
- **Application Manager**. Provides operations to inform applications about the results of pending requests such as allocation of flows or registrations of applications.

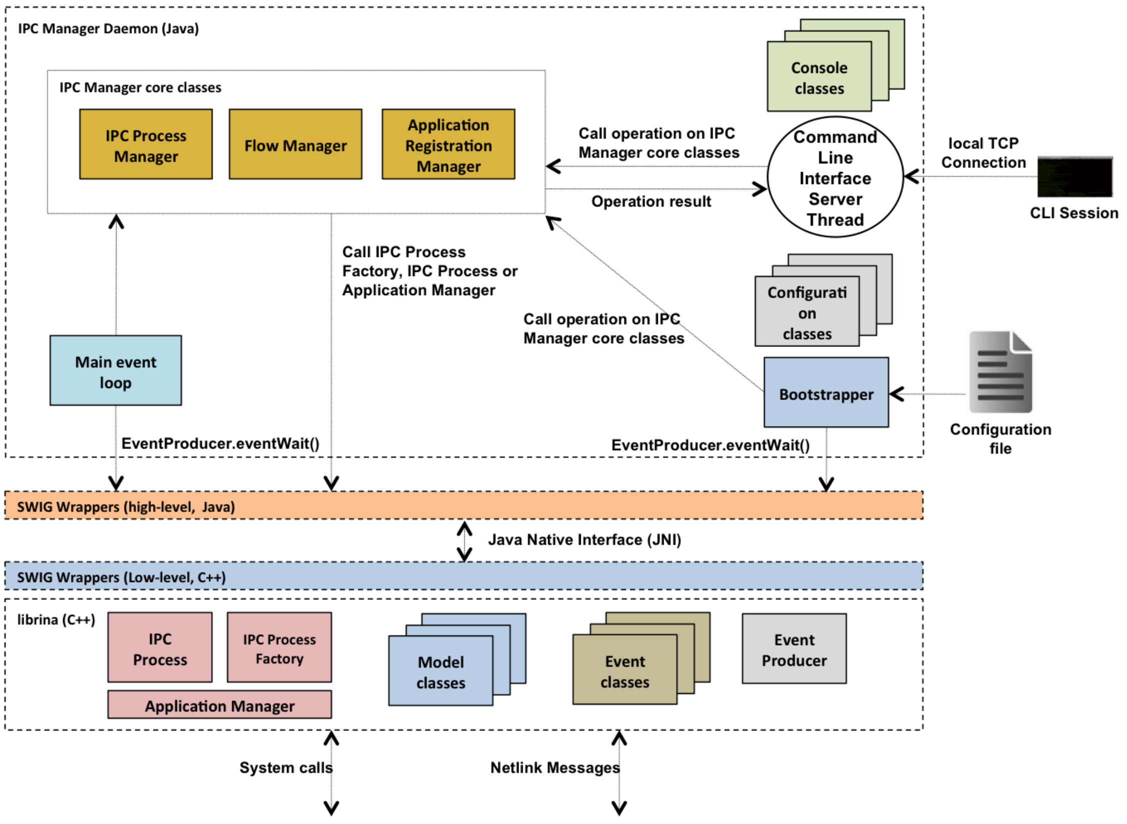
IPC Manager Daemon (Java)
IPC Manager core classes
IPC Process Manager
Flow Manager
Application Registration Manager
Console classes
Call operation on IPC Manager core classes
Operation result
Command Line Interface Server Thread
local TCP Connection
CLI Session
Call IPC Process Factory, IPC Process or Application Manager
Call operation on IPC Manager core classes
Configuration classes
Bootstrapper
Configuration file
Main event loop
EventProducer.eventWait()
EventProducer.eventWait()
SWIG Wrappers (high-level, Java)
Java Native Interface (JNI)
SWIG Wrappers (Low-level, C++)
librina (C++)
IPC Process
IPC Process Factory
Application Manager
Model classes
Event classes
Event Producer
System calls
Netlink Messages

Figure 20 IPC Manager Daemon detailed software design

When the IPC Manager Daemon initializes it reads a configuration file from a well-known location. This configuration file provides default values for system parameters, describes configurations of well-known DIFs and controls the behaviour of the IPC Manager bootstrap process. The latter is achieved by specifying:

- The IPC Processes that have to be created at system start-up, including their name and type.
- For each IPC Process to be created, the names of the N-1 DIFs where the IPC Process has to be registered (if any).
- For each IPC Process to be created, the name of the DIF that the IPC Process is a member of (if any). If the IPC Process is assigned to a DIF it will be initialized with an address and all the other information required to start operating as a member of that DIF (DIF-wide constants, policies, credentials, etc.)

When the bootstrapping phase is over the IPC Manager main thread starts executing the event loop forever. The event loop continuously polls librina's EventProducer (in blocking mode) to get the events resulting from Netlink request messages sent by applications or IPC Processes.

When and event happens, the event loop checks its type and delegates the processing of the event to one of the specialized core classes: Flow Manager (flow related events), Application Registration Manager (application-registration related events) or IPC Process Manager (IPC Process lifecycle management related events). The processing performed by these core classes will typically result in the invocation of one of the operations provided by the `librina-ipc-process` Proxy classes previously described in this section.

Local system administrators can interact with the IPC Manager through a Command Line Interface (CLI), accessible via `telnet`. This console provides a number of commands that allow system administrators to query the status of the RINA stack in the system, as well as performing actions that modify its configuration (such as creating/destroying IPC Processes, assigning them to DIFs, etc.). The IPC Manager supports the CLI console through a dedicated thread that listens at the console port; only one console session at a time is supported at the moment.

The current IPC Manager has leveraged the following Alba components, adapting them to the environment of the IRATI stack:

- Configuration file format, parsing libraries and model classes (the configuration file uses JSON – the JavaScript Object Notation).
- Command Line Interface Server Thread and related parsing classes.
- Bootstrapping process.

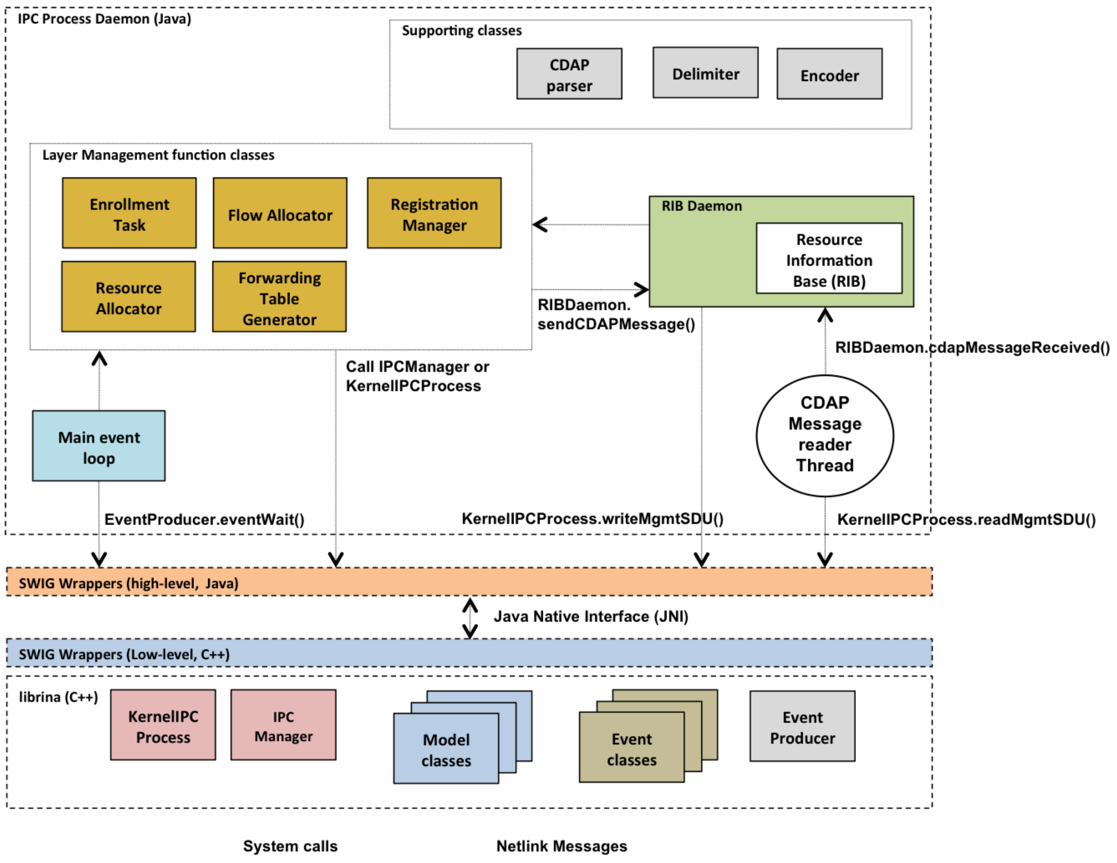
IPC Process Daemon (Java)
Supporting classes
CDAP parser
Delimiter
Encoder
Layer Management function classes
Enrollment Task
Flow Allocator
Registration Manager
Resource Allocator
Forwarding Table Generator
RIB Daemon
Resource Information Base (RIB)
RIBDaemon. sendCDAPMessage()
RIBDaemon.cdapMessageReceived()
Call IPCManager or KernelIPCProcess
Main event loop
CDAP Message reader Thread
EventProducer.eventWait()
KernelIPCProcess.writeMgmtSDU()
KernelIPCProcess.readMgmtSDU()
SWIG Wrappers (high-level, Java)
Java Native Interface (JNI)
SWIG Wrappers (Low-level, C++)
librina (C++)
KernelIPC Process
IPC Manager
Model classes
Event classes
Event Producer
System calls
Netlink Messages

Figure 21 IPC Process Daemon detailed software design

The IPC Process Daemon performs the layer management functions of a single IPC Process. It is therefore “half” of the IPC Process application, while the other half – dealing with data-transfer and data-transfer control related tasks - is located at the kernel. Layer management operations are more complex and do not have such stringent performance requirements as data transfer operations, therefore locating them at user-space is a logical choice, as introduced in D2.1.

Figure 21 depicts the detailed software design of the IPC Process Daemon. The first phase prototype follows the same approach taken with the IPC Manager Daemon design and implementation: leveraging the Alba stack as much as possible in order to provide a simple but complete enough implementation of the IPC Process Daemon. Therefore the IPC Process Daemon is also a Java OS process that builds on the APIs exposed by librina through SWIG and JNI. The librina proxy classes described below are the more relevant to the IPC Process Daemon operation:

- **IPC Manager**. Allows the IPC Process Daemon to communicate with the IPC Manager Daemon, mainly to inform the latter about the results of requested operations; but also to notify about incoming flow requests or flows that have been deallocated.
- **Kernel IPC Process**. Provides operations to enable the IPC Process Daemon to communicate with the data-transfer/data-transfer-control related functions of the IPC Process in the kernel. The APIs allow the IPC Process Daemon to modify the kernel IPC Process configuration, to manage the setup and teardown EFCP connections or to modify the PDU forwarding table.

IPC Process Daemons are instantiated and destroyed by the IPC Manager Daemon. When the IPC Process Daemon has completed is initialization, the main thread starts executing the event loop. Such a loop is implemented by continuously polling the EventProducer for new events (in blocking mode) and processing them when they arrive. The event processing is delegated to the classes implementing the different layer management functions: Enrollment Task, Resource Allocator, Registration Manager, Flow Allocator and PDU Forwarding Table Generator. Processing performed by these classes typically involves two types of actions:

- Local actions resulting in communications with the Kernel IPC Process or the IPC Manager, achieved via the librina proxy classes.
- Remote actions resulting in communications with peer IPC Process Daemons, achieved via the RIB Daemon.

The RIB Daemon is an internal component of the IPC Process that provides an abstract, object-oriented schema of all the IPC Process state information. This schema, known as the Resource Information Base or RIB, allows IPC Processes to modify the state of their peers by performing operations on one or more of the RIB objects. The Common Distributed Application Protocol (CDAP) is the application protocol used to exchange the remote RIB operation requests and responses between peer IPC Processes. This protocol allows six remote operations to be performed over RIB objects: create, delete, read, write, start and stop. The objects that are the target of the operation are identified by the following attributes:

- **Object class**. Uniquely identifies a certain type of objects.
- **Object name**. Uniquely identifies the instance of an object of a certain class. The object class + object name tuple uniquely identify an object within the RIB.
- **Object instance**. A shorthand for object class + object name, to uniquely identify an object within the RIB.
- **Scope**. Indicates the number of 'levels' of the RIB affected by the operation, starting at the specified object (object class + name or instance). This allows a single operation to target multiple objects at once.

- **Filter**. Provides a predicate that evaluates to 'true' or 'false' based on the value of the object attributes. This allows further discriminating to what objects the operation has to be applied.

More information about the RIB, RIB Daemon and CDAP can be found at D2.1 [3].

CDAP is implemented as a library that provides a `CDAPSessionManager` class that manages one or more CDAP sessions. The `CDAPSession` class implements the logics of the CDAP Protocol state machine as defined in the CDAP specification [33]. CDAP can be encoded in multiple ways, but the IRATI stack follows the approach adopted by the other current RINA protocols to use Google Protocol Buffers (GPB) [31]. This decision will make interoperability possible, and will also provide the benefits of GPB: efficient encoding; proven, mature and scalable technology with good quality open source parsers/generators available.

In addition to the information of the operation as well as the identity of the targeted objects, CDAP messages can also transport the actual values of such objects. Therefore the object values also need to be encoded in binary format. Again, GPB is the initial encoding format chosen, although others are also possible (ASN.1, XML, JSON, etc). Object encoding functionalities are implemented by the `Encoding` support library, which provides an encoding-format-neutral interface. Thus it allows for several encoding implementations to be plugged in/out, specifying which one to use at configuration time.

The RIB is implemented as a map of object managers, indexed by object names – current RINA implementations have adopted the convention of making object names unique within the RIB as a simplifying assumption. Each object manager wraps a piece of state information (for example Flows, Application Registrations, QoS Cubes, the PDU Forwarding Table, etc) with the `RIBObject` interface. This interface abstracts the six operations provided by CDAP: create, delete, read, write, start and stop. When a remote CDAP message reaches the IPC Process Daemon, the message is handled to the RIB Daemon component. The RIB Daemon retrieves the object manager associated to the targeted object name from the RIB map, and invokes the requested CDAP operation. The goal of the object manager is to translate each CDAP operation to the appropriate actions on the layer management function classes.

The layer management function classes use the RIB Daemon when they have to invoke a remote operation to a peer IPC Process. The RIB Daemon provides operations to send CDAP messages to neighbour IPC Processes based on its application process name. When such operations are called, the RIB Daemon internally fetches the port-id of the underlying N-1 flow that allows the IPC Process to communicate with the given neighbour, encodes the CDAP message and requests the kernel to write the encoded CDAP message as an SDU to that N-1 flow.

The current IPC Process Daemon has leveraged the following Alba components, adapting them to the environment of the IRATI stack:

- Supporting classes: CDAP library, Encoder library.
- RIB Daemon and RIB implementation classes.
- Layer management function classes: Enrolment Task, Resource Allocator, Flow Allocator and Registration Manager.

The PDU Forwarding Table Generator implementing the specification of D2.1 is not part of Alba. Currently the implementation of this component is in progress and will be part of prototype 2 (D3.2).

2.3 Package inter-dependencies

The IRATI stack is now split into three different software packages: the kernel, librina and rinad. This partitioning allows IRATI adopters to pick up and import only the parts they require into their solutions with less effort compared to monolithic package solutions.

The following figure shows the inter-packages run-time dependencies.

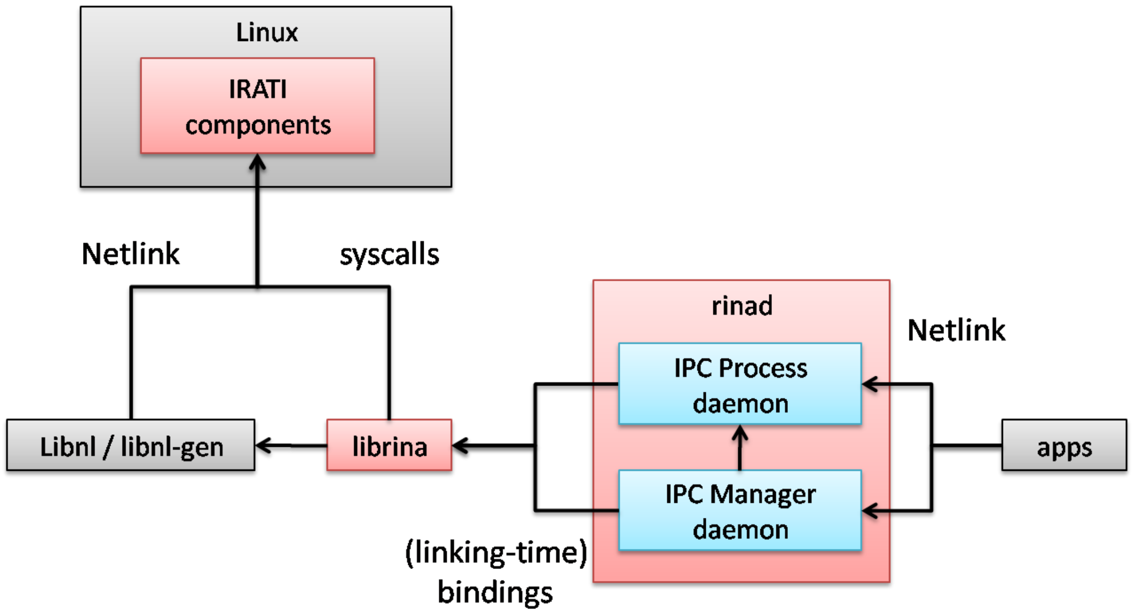

Figure 22 Packages runtime inter-dependencies

The building dependencies among the packages (e.g. rinad depends on the availability of librina already installed into the system) are maintained and checked by each package configuration framework (i.e. autotools). Some of the features the packages provide are optional and can be explicitly disabled at configuration time. Moreover, the configuration system automatically adapts to systems with reduced (optional) requirements.

The following figure shows the packages build-time first-level inter-dependencies. Refer to the documentation accompanying the sources (e.g. README files) for further details.

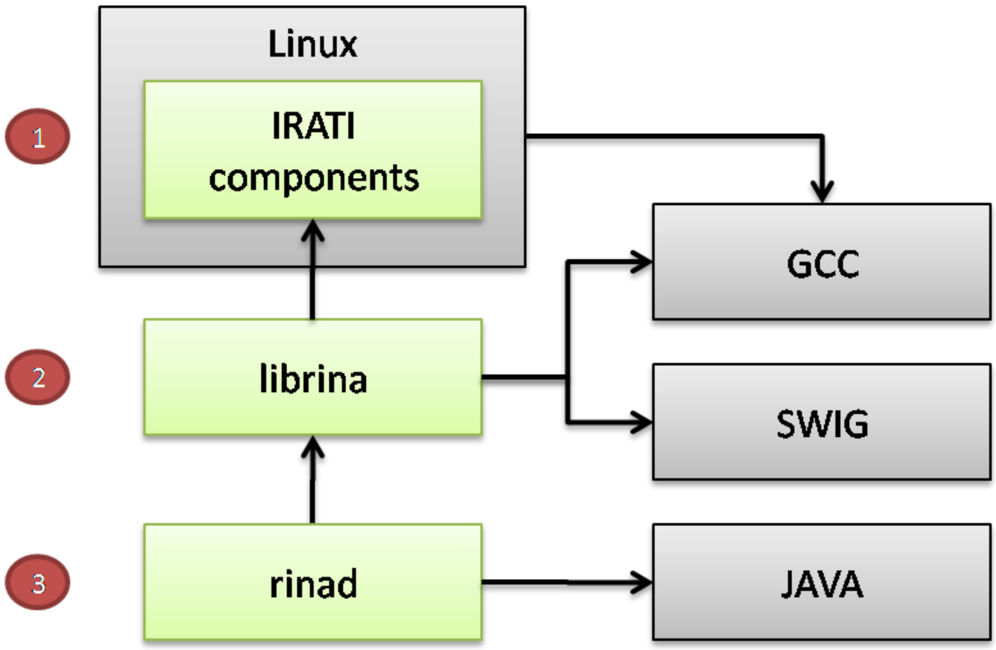

Figure 23 Packages build-time inter-dependencies

The auto-configuration features described in section 5.2 and 5.3 have been introduced into the user-space packages (i.e. librina and rinad) to allow compiling them into different OS/Linux systems. The packages can be built into reasonably updated systems different than the one selected by IRATI Project partners for the development and integration activities (i.e. Debian/Linux).

3 The development and release model

The whole development of the IRATI stack is managed with the git SCM [5]. Git is a free and open source distributed version control system (DVCS) designed to handle everything from small to very large projects with speed and efficiency. Please refer to [6] for further information.

The IRATI project git repository is currently hosted on a private repository at github [7]. The repository will not be opened up until prototype 2 (D3.2) is delivered, since the IRATI consortium wants to guarantee minimal standards of quality and a certain stability of the APIs in order to make the prototype usable by the interested stakeholders.

In order to get a copy of the repository, the following command must be issued:

```
git clone git@github.com:dana-i2cat/irati.git foo
```

Upon successful completion, the `foo` directory will be holding a local copy of the project source base as well as the complete commits history since the developments very beginning. The ‘`git log`’ command or Graphical User Interfaces (GUI) programs, such as qgit [10] or gitk [9], can be used to navigate the commits history, retrieve logs, tags information etc.

3.1 The git workflow

The git workflow agreed among the partners varies widely depending on different conditions such as the deadlines prioritization, the concomitant introduction of new features and stabilization fixes, temporary workarounds etc. In the following sections the usual git workflow is described.

3.1.1 Repository branches

The development, integration and testing procedure shared among the partners mostly pivot around the following repository branches:

- *wip* (work in progress): This branch holds the in-progress work. Such code can have temporary workarounds, fixes and partial implementations etc. The *wip* branch is used to integrate all the partner contributions and, once declared sufficiently stable, gets merged into the *irati* branch.
- *irati*: This branch holds the “stable” IRATI software. The *master* branch sources are merged into *irati* as well as all the partner contributions. After passing the integration tests, the *irati* branch software is finally released to WP4.
- *master*: this branch is used to hold the third-parties (mainstream) sources as they are, e.g. the vanilla Linux sources.

An example of the usual approach can be summarized with the following events:

- The partners agree on importing a new mainstream release (e.g. for bug-fixes or stabilization): the mainstream source code is integrated into the *master* branch which successively gets propagated (merged) into the *irati* branch and then into *wip* respectively. For example, a notable mainstream update can be represented by the introduction of a new Linux kernel version.
- A bunch of contributions in *wip* is declared stable: the *wip* branch is merged into *irati*
- Important fixes have been indicated by WP4 testing activities: the fixes are applied, as point-solutions, into the *irati* branch which successively gets merged into *wip*

3.1.2 Internal software releases

Due to the cooperative nature of the stack' components and the deployment constraints in the project' testbed, work-packages WP3 and WP4 agreed on exchanging software releases directly through tagged repository versions. Such tags have the following format:

v<**MAJOR**>.<**MINOR**>.<**MICRO**>

where:

- `MAJOR`: is incremented when the software base reaches a mature level of stability
- `MINOR`: is incremented when all the planned functionalities introduced have been integrated, tested and the resulting prototype is sufficiently stable to be used without major faults or crashes.
- `MICRO`: is incremented when a set of contributions (representing a new functionality introduced) gets merged into the *wip* branch

Each tag also reports a message briefly describing the changes involved into the release (which can be analyzed in details through the use of '`git log`' command).

e.g.:

```
git tag -l -n1 | sort -V
...
v0.1.9   Version bump (fullchain)
v0.2.0   Version bump (demo ready)
...
v0.2.5   Version bump (arp826, rinarp and shim-eth-vlan
fixes)
...
v0.4.11  Version bump (user-space fixes)
v0.4.12  Version bump (IPCP-loop ready)
```

The whole source code base, referring to a particular tag, can be accessed with the following command:

```
    git checkout <tag>
```

The tag-based release approach allows releasing software snapshots to WP4 often at the minimum expense in terms of time and efforts.

3.1.3 Issues management

The github issue and repository tracking services [19] allow to bind Software Problem Reports (SPR), such as feature requests or bug reports, to milestones as well as to automatically refer to SPR from commits. The features and the automatisms available brought the partners to embrace the github services for managing all the IRATI software.

4 The environments

In the following sections the kernel and user spaces building, development and testing environments are described.

4.1 The build environments

4.1.1 The kernel-space configuration and building

The Linux kernel has two ad-hoc systems for configuration and building purposes: the Kconfig and Kbuild frameworks respectively.

The Kconfig framework [1] allows selecting kernel compilation-time features with a User Interface (UI) that drives the selections and automatically enables/disables dependant configuration entries (e.g the IPv4 support can be selected only if the network support has been previously enabled, the FS support is automatically enabled by default etc.).

The Kbuild framework [2] is a make [17] wrapper specifically tailored for building the kernel image file and dynamically loadable modules, handling all the particularities of their details (e.g. embedded linking scripts, modules symbols mangling, firmware embedding, dynamic modules generation, syscalls table generation, source files auto-generation, static stack checks).

The kernel-space parts of the IRATI stack are contained into the `linux/net/rina` repository path; which holds the source code, the Kconfig file(s) and the Kbuild file(s).

The current build setup provides to the final user the following dynamically loadable kernel modules:

Module name	Description	Pre-requisite
rina-personality-default	The IRATI personality. It holds all the base components of the stack such as the framework libraries, RNL, IPCP Factories, EFCP, RMT PDU-FWD-T, KIPCM, KFA etc.	-
normal-ipcp	The Normal IPC Process	rina-personality-default
shim-dummy	The Shim Dummy IPC Process	rina-personality-default
shim-eth-vlan	The Shim Ethernet IPC Process	rina-personality-default, rinarp
shim-tcp-udp	The Shim TCP/UDP IPC Process	rina-personality-default
rinarp	The RINARP component	arp826
arp826	The ARP826 component	-

The modular approach can be summarized as in the following steps:

- Upon start-up/loading:
    1. The kernel image is loaded: The IRATI stack core, which is built-in into the kernel image, initializes the framework, the KIPCM, the KFA and the RNL layer. Please note that the stack core cannot be a dynamically loadable module since it has to publish the RINA system calls into the global syscalls table. The syscalls table is static, produced at compilation time.
    2. Upon `rina-personality-default` module loading: the personality hooks get registered into the core.
    3. Upon IPC Process module loading (i.e. `normal-ipcp`, `shim-dummy` or `shim-eth-vlan`): the module registers its hooks to the IPC Process Factory.
- Upon shutdown/unloading
    4. Whenever an IPC Process module is explicitly unloaded, the module deregisters as an IPC Process Factory from the system
    5. Once all the IPC Process Factories are removed, the `rina-personality-default` module can be unloaded from the system.

The following figure depicts the aforementioned steps

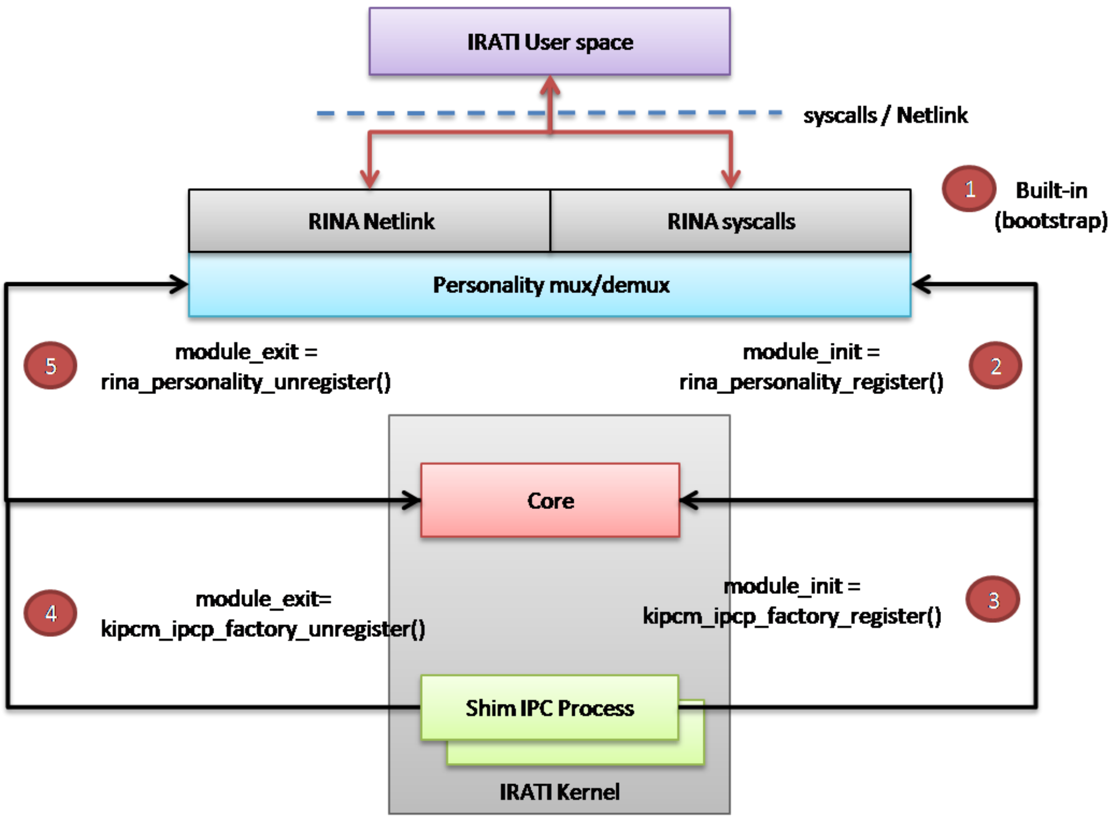
IRATI User space
syscalls / Netlink
1
Built-in (bootstrap)
RINA Netlink
RINA syscalls
Personality mux/demux
5
module_exit = rina_personality_unregister()
module_init = rina_personality_register()
2
Core
4
module_exit= kipcm_ipcp_factory_unregister()
module_init = kipcm_ipcp_factory_register()
3
Shim IPC Process
IRATI Kernel

Figure 24 The module loading/unloading steps

4.1.2 The user-space configuration and building

All the IRATI stack packages in user-space share the same configuration and building frameworks. These frameworks are built with the GNU build system, also known as the "autotools" suite.

The autotools suite is a set of development tools designed to assist developers and package maintainers in making source-code packages portable to many Unix-like systems. The suite is designed to make it possible to build and install a software package by running the following shell commands:

```
./configure && make && make install
```

The autotools suite can be summarized as composed by the following tools:

- *Autoconf*: Autoconf [13] is an extensible package that produces POSIX shell scripts to automatically configure software source code packages. These scripts can adapt the packages to many kinds of UNIX-like systems without manual user intervention. Autoconf creates a configuration script (i.e. `configure`) from a template file that lists the operating system features that the package can use.
- *Automake*: Automake [14] is a tool for automatically generating Makefiles. Each automake input file is basically a series of make [17] variable definitions, with rules being thrown in occasionally. The goal of Automake is to remove the burden of Makefiles maintenance.
- *Libtool*: Libtool [15] simplifies the shared libraries generation procedure by encapsulating both the platform-specific dependencies, and the library user interface, in a single point. Libtool is designed so that the complete functionality of each host type is available via a generic interface while the various systems quirks are hidden to the developer.
- *pkg-config*: Pkg-config [16] is a helper tool that provides a unified interface for querying installed libraries, mainly for the purpose of compiling software from its source code. It simplifies the development of the autoconf checks, which are performed at configuration-time. The tool is language-agnostic, therefore it can be used for defining the location of documentation, interpreted language modules, configuration files etc.

Please refer to each package's documentation [13-16] for further details.

4.2 The development and testing environments

The WP3 development and testing environment constitutes of a Debian-testing (currently codenamed "Jessie") based VM image (VMI).

The VMI is updated upon the introduction of new development or testing requirements (i.e. external tools) and shared among the partners through the project FTP Server [12]. All the VMIs contain the pre-requisite tools required to build and test the stack.

The partners agreed on using VirtualBox [21] as the virtualization solution for running the IRATI Virtual Appliances (VA) since it allows to easily setup environments that suit the WP3 integration testing scopes. The IRATI Stack requirements do not imply ad-hoc virtualization architectures and therefore other virtualization facilities can be applied.

5 Installing the software

As previously described, the IRATI stack is composed by the Linux kernel, the librina package and the rinad package. In the following sections, specific per-package configuration and building instructions are described.

5.1 Installing the kernel

The configuration and building system of the IRATI stack kernel-space parts strictly follows the Linux system. Therefore, the following steps – as well as almost all the documentation freely available on the World Wide Web - can be followed.

From the `linux` root folder:

1. Either
   a. Copy the `config-IRATI` template file onto the default configuration file (i.e. '`cp config-IRATI .config`').
   b. Configure the kernel with the UI based tool (i.e. '`make menuconfig`').
2. Type '`make bzImage modules`' in order to build the kernel and all the modules.
3. Install the kernel and all the modules in the target system by typing '`make modules_install install`'.

Please note that OS/Linux distributions do not usually require additional bootloader setup and the '`make module_install install`' command is enough to a) install the new kernel and b) setup the bootloader in order to allow booting the kernel. However, the bootloader configuration procedure may change and different procedures may have to be followed. Please refer to the distribution documentation for specific instructions and eventual troubleshooting.

5.2 Installing the librina package

The following sections summarize the steps required for installing librina from a repository checkout.

5.2.1 Bootstrapping the package

The sources, as cloned from the repository, do not contain the necessary setup that would allow a successful package configuration. In order to setup the sources environment, run the '`bootstrap`' script available in the package root as follows:

```
./bootstrap
```

5.2.2 Prerequisites for the configuration

The librina package requires the following external packages (already installed into the development VMI) in order to complete the configuration process successfully:

1. A C++ compiler
2. libnl-3
3. libnl-3-genl
4. swig >= 2.0 (excluding versions in the range [2.0.4, 2.0.8])

5.2.3 Installation procedure

Once the package has been "bootstrapped", as described in section 5.2.1, the usual "autotools" procedure must be followed to configure, build and install the package:

```
1. ./configure
2. make
3. make install
```

Please refer to the `configure` script help for further details on the command line options available. The `configure` help can be obtained with the following command:

```
./configure –help
```

The configure script assumes default parameters if no options are given. The package installation paths are set to the system defaults. If the paths have to be changed, in order to suit a different installation, the `--prefix` configure option can be used as follows:

```
./configure --prefix=<PREFIX>
```

5.3 Installing the rinad package

The following sections summarize the steps required for installing rinad from a repository checkout.

5.3.1 Bootstrapping the package

The sources, as cloned from the repository, do not contain the necessary setup that would allow a successful package configuration. In order to setup the sources environment, run the '`bootstrap`' script available in the package root as follows:

```
./bootstrap
```

5.3.2 Prerequisites for the configuration

The rinad package requires the following external packages (already installed into the development VMI) in order to complete the configuration process successfully:

1. librina
2. The Java virtual machine (JRE >= 6)
3. The Java compiler (javac)
4. Maven

5.3.3 Installation procedure

Since rinad relies on the librina wrappers availability (i.e. the shared librina low-level wrappers libraries as well as the Java high-level JAR wrappers), the configure script has to be instructed on the paths where to look for the wrappers. With reference to the `PREFIX` variable value used during the librina configuration process (as described in section 5.2.3), the following commands allows configuring the rinad package:

```
     1.     PKG_CONFIG_PATH=$PREFIX/lib/pkgconfig
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PREFIX/lib ./configure --
prefix=$PREFIX
     2.     make
     3.     make install
```

Please note that the rinad package relies on Maven to compile all the Java parts and, as a side effect, Maven will be downloading all its prerequisites during the execution of the ‘`make`’ command (ref. section 5.2.3, step #2).

5.4 Helper for librina and rinad installation

The repository source directory contains a couple of helper scripts summarizing the installation and removal procedures of the user space IRATI stack packages:

- `install-from-scratch`: this script installs both librina and rinad packages. The script takes, as an optional parameter, the absolute path of the installation (i.e. the package `PREFIX`).
- `uninstall-and-clean`: this script is the opposite of `install-from-scratch`, it uninstalls all the software previously installed and clean ups the source directories.

5.5 Layout of folders and files in the installation directory

With reference to the `PREFIX` variable used during the installation, the following table describes the paths utilized during the user-space packages installation procedure.

Path	Description
`bin`	Holds the different binaries produced by the packages, e.g. `echo-client`, `echo-server`, `ipcmanager`, `ipcprocess`, `rinaband-client` and `rinaband-server`
`etc`	Holds the configuration files used by the different programs (e.g. the IPC Manager configuration file `ipcmanager.conf`)
`include/librina`	Holds all the header files of librina
`lib`	Contains the librina static/shared libraries
`lib/pkg-config`	Holds the pkg-config data files
`share/librina`	Contains the JARs of the librina Java wrappers
`share/rinad`	Contains the JARs required to run the various daemons (i.e. IPC Manager and IPC Process) and applications (i.e. RINABand and Echo applications)

6 Loading and running the stack

At the end of the bootstrap procedure, the following modules have to be loaded, in order to get a fully functional IRATI stack:

```
1. modprobe rina-personality-default
2. modprobe shim-eth-vlan
3. modprobe normal-ipcp
```

The aforementioned steps automatically load all the necessary prerequisite modules, as described in section 4.1.1 (e.g. the loading of the shim-eth-vlan module triggers the loading of the rinarp modules which in turn will cause the loading of the arp826 module).

Please note that if the kernel is a non-modular one, the aforementioned commands are unnecessary. All the IRATI kernel components will be built-in into the kernel image and initialized during the system bootstrap phase.

In order to have a functional stack, the remaining IPC Manager and IPC Process daemons have to be executed. Since the IPC Manager automatically starts a new IPC Process when needed, the only remaining command that has to be executed is the following:

```
1. $PREFIX/bin/ipcmanager
```

Please note that the IPC Manager daemon relies on a configuration script, located into `$PREFIX/etc` path, which could have to be changed in order to suit specific needs.

7 Testing the stack

With reference to a VirtualBox [11] based environment, the following instructions can be used to setup an environment suitable for testing the prototype between two Virtual Machines (VMs) intercommunicating through a VLAN, as depicted in the following figure:

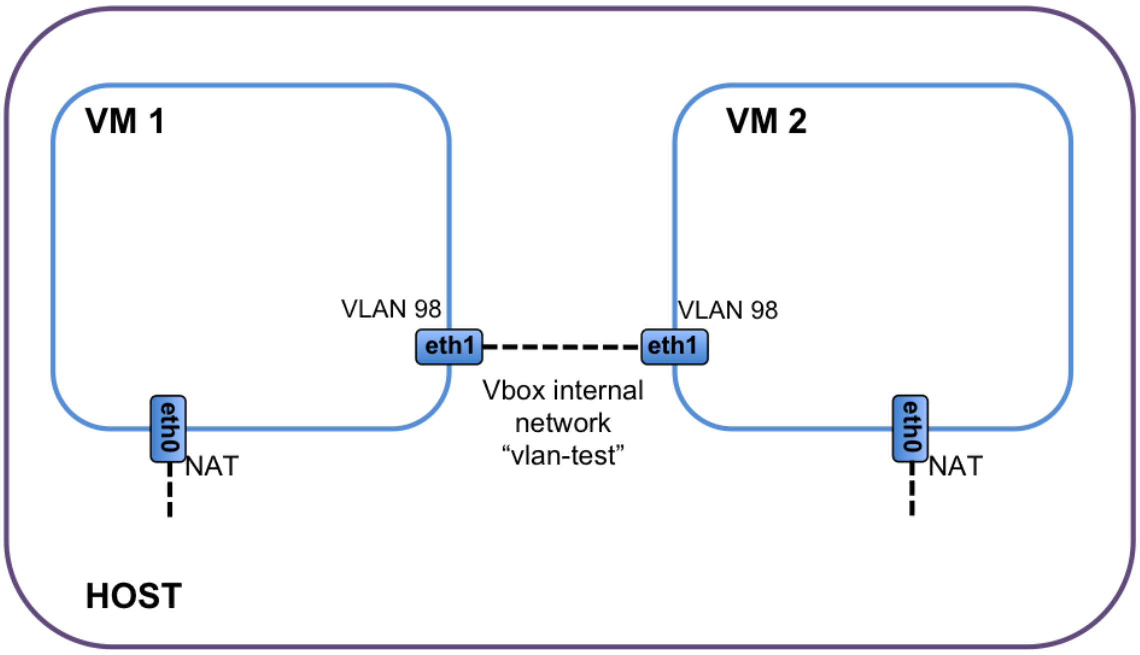

Figure 25 Testing environment

Please note that the instructions are general and can be used in a real HW deployment, with just the minimum amount of changes required to obtain the correct network setup.

7.1 VirtualBox configuration

The two VMs must have two NICs each. The first adapter will be used for accessing the outside world (e.g. for the Maven prerequisites retrieval during the rinad package installation) while the second will be used for testing the stack, using the Shim Ethernet IPC Process.

Therefore, the VirtualBox configuration for each VM will be assumed as the following:

- eth0: A NAT-ed interface, with access to the Internet through the physical host.
- eth1: An interface directly connected to the Virtual Box internal network. The VLAN will be configured on this interface.

The VM Images (VMIs) available for download from [12] and the instructions presented in sections 5 and 6 can be used to setup the testing environment. Please refer to [21] for instructions on the VMs NICs setup.

7.2 VMs configuration

In order to properly configure the guest OS with the setup required for the test, the following steps can be followed in one VM:

1. Enable the 8021q module (i.e. `modprobe 8021q`).
2. Add VLAN 98 to interface eth1 (i.e. `vconfig add eth1 98`).
3. Enable and configure eth1.98:
   3.1 `ifconfig eth1.98 up`
   3.2 `ifconfig eth1.98 mtu 1496` (optional if the network card does not support 1504B MTU)

The other VM configuration will be the same.

7.3 IPC Manager configuration file

In order to facilitate the testing, the IPC Manager can be configured to automatically create a shim-Ethernet over VLAN IPC Process and assign it to a DIF. This way, applications can start using it as soon as the IPC Manager completes its initialization.

The aforementioned feature can be enabled by editing the IPC Manager configuration file (ref. section 5.5) as follows:

```
{
  "localConfiguration" : {
    "installationPath" : "/usr/local/irati/share/rinad",
    "libraryPath"      : "/usr/local/irati/lib",
    "consolePort"      : 32766
  },
  "ipcProcessesToCreate" : [ {
    "type"                       : "shim-eth-vlan",
    "applicationProcessName"     : "test",
    "applicationProcessInstance" : "1",
    "difName"                    : "98"
  } ],
  "difConfigurations" : [ {
    "difName"          : "98",
    "configParameters" : [ {
        "name"  : "interface-name",
        "value" : "eth1"
      } ]
  } ]
}
```

This configuration file will instruct the IPC Manager to create a Shim Ethernet IPC Process with application process name “`test`”, application process instance “`1`” and assign it to the DIF “98” (ref. `ipcProcessToCreate`). The configuration of the DIF “98” is described below (ref. `difConfigurations`), with a single configuration parameter to bind the shim IPC Process to the “`eth1`” network interface (ref. `configParameters`).

7.4 Running the test

Depending on the `PREFIX` path used during installation (ref. section 5.2 and 5.3), the IRATI stack binaries will be located in a certain directory. In the following, the `IRATI_BIN` shell variable value is assumed to be set to `$PREFIX/bin`.

1. Start the IPC Manager: `$IRATI_BIN/ipcmanager`
2. On VM1:
    a. Start the “echo application” in server mode: `$IRATI_BIN/echo-server`
3. On VM2:
    a. Start the “echo application” in client mode: `$IRATI_BIN/echo-client`

Please note that the IPC Manager daemon automatically starts instances of the IPC Process daemon thus, there is no need to start the IPC Process daemon manually.

The `echo-client` and `echo-server` will start exchanging SDUs over eth1.98. Use a traffic sniffer such as tcpdump [24] or WireShark [25] to analyze the RINA related traffic flowing through eth1.98.

8 Conclusions and future work

The software components required for the first phase IRATI project prototype have been designed, developed, integrated and functionally tested.

Although the design and development activities progressed without major problems and deviations, the unsuitability of the Linux ARP implementation - for the scope of the Shim Ethernet IPC Process - introduced unplanned developments, slightly delaying the prototype release date. These delays coupled with the project' tight timings induced prioritization among the features to be delivered in the first phase. The prototype now implements core parts of a RINA stack over Ethernet for a Linux-based OS, provides a solid backbone for the upcoming developments and supports the creation of DIFs over Ethernet with levels of service similar to UDP. Its DTCP implementation is under active development while placeholders within CDAP have been defined and authentication mechanisms are going to be added in short. Routing and multi-homing support has been post-poned for inclusion in the phase 2 prototype, in order to stabilise the groundworks first.

The software prototype meets both the features and the stability requirements for experimentation and has been delivered to WP4 for experimentations. Results of the experimentation phase are described in deliverable D4.1 [30].

The core functionalities described in this document will be further enhanced and the current prototype will be incrementally updated with additional features towards fulfilling the project' goals. Apart from these planned features, the partners agreed on a set of improvements to be applied to the source code base. These enhancements can be summarized as follows:

- Incrementally redesign the ingress/egress DU loop, in order to obtain a distributed architecture that does not incur in the bottle-necks caused by the current centralized DU-loop approach (i.e. the KIPCM and KFA sustain the DUs ingress and egress loop, as described in section 2.1.5).
- Reduce the kernel-space memory footprint, by embracing techniques such as flight-weight objects, Copy-On-Write (COW) memory optimization strategies, objects caching etc.
- Provide an improved sysfs integration, in order to properly expose the stack' objects internals to the user. This would enable for greater interaction allowing either to read the objects status or to change their exported parameters while the stack is running (i.e. dynamically profiling the stack while it is running).
- Improve the user-space performance by translating the IPC Process and IPC Manager daemons from Java to C++.
- Improve the stack' POSIX compliance and decouple even more the kernel-space components from the underlying kernel functionalities/libraries, in order to ease

porting the stack to other POSIX compliant systems (Debian GNU/kFreeBSD, FreeBSD etc.).

The updated prototypes will be made available in the next periods as part of the upcoming project deliverables D3.2 and D3.3.

9 References

1. The Linux Kbuild building framework – https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt
2. The Linux Kconfig configuration framework https://www.kernel.org/doc/Documentation/kbuild/kconfig.txt
3. IRATI, Deliverable D2.1 – http://irati.eu/wp-content/uploads/2012/07/IRATI-D2.1.pdf
4. RFC 3549 Linux Netlink as an IP Services Protocol - http://www.ietf.org/rfc/rfc3549.txt
5. git – http://git-scm.com/
6. git documentation – http://git-scm.com/documentation
7. github – https://github.com
8. The IRATI software repository (access available on demand) – https://github.com/dana-i2cat/irati
9. gitk: The git repository browser – https://www.kernel.org/pub/software/scm/git/docs/gitk.html
10. qgit: A graphical interface to git repositories – http://sourceforge.net/projects/qgit
11. Virtual Box – https://www.virtualbox.org
12. IRATI FTP Server (access available on demand) – ftp.i2cat.org
13. GNU autoconf – http://www.gnu.org/software/autoconf
14. GNU automake – http://www.gnu.org/software/automake
15. GNU libtool – http://www.gnu.org/software/libtool
16. pkg-config – http://www.freedesktop.org/wiki/Software/pkg-config
17. GNU Make – http://www.gnu.org/software/make
18. SWIG: The Software Wrapper and Interface Generator – http://swig.org
19. The IRATI issues tracking system (access available on demand) – https://github.com/dana-i2cat/irati/issues
20. The Alba RINA stack – https://github.com/dana-i2cat/alba
21. The VirtualBox user manual – https://www.virtualbox.org/manual/UserManual.html
22. The Python API – http://docs.python.org/2/extending/extending.html
23. The Java Native Interface – http://docs.oracle.com/javase/7/docs/technotes/guides/jni
24. tcpdup – http://www.tcpdump.org
25. WireShark – http://www.wireshark.org
26. RFC-826, An Ethernet Address Resolution Protocol – http://tools.ietf.org/search/rfc826
27. The zen of kobjects – http://lwn.net/Articles/51437
28. A. T. Schreiner, "Object oriented programming with ANSI-C," 1993 – https://ritdml.rit.edu/handle/1850/8544
29. Opaque pointers – http://en.wikipedia.org/wiki/Opaque_pointer
30. IRATI, Deliverable D4.1 – http://irati.eu/wp-content/uploads/2012/07/IRATI-D4.1-final.pdf

31. Google Protocol Buffers Developers Guide - https://developers.google.com/protocol-buffers/
32. Linux SysFS - https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt
33. CDAP - Common Distributed Application Protocol Reference (available on demand) - PNA Technical Draft D-Base-2010-xxy, draft 0.7.2, December 2010

END OF DOCUMENT